# A Compositional Approach to Embedded System Design

Bei der Gemeinsamen Fakultät für Maschinenbau und Elektrotechnik
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung der Würde

eines Doktor-Ingenieurs (Dr.-Ing.)

genehmigte Dissertation


| | |
|---|---|
| von: | Dirk Ziegenbein |
| aus: | Braunschweig |



| | |
|---|---|
| eingereicht am: | 28. Januar 2002 |
| mündliche Prüfung am: | 13. März 2002 |
| Berichterstatter: | Prof. Dr.-Ing. Rolf Ernst |
| | Prof. Dr.-Ing. habil. Lothar Thiele |

2003

# Abstract

An important observable trend in embedded system design is the growing system complexity. Fueled by increasing device integration capabilities of semiconductor technology, more functions can be implemented at the same cost. Besides the sheer increase of the problem size, the growing complexity has another dimension which is the resulting heterogeneity with respect to the different functions and components of an embedded system. This means that functions from different application domains are tightly coupled in a single embedded system. For example, a mobile phone has reactive functions (protocol processing), transformative functions (signal processing), and interactive functions (user interface). It is established industry practice that specialized specification languages and design environments are used in each application domain. The resulting heterogeneity of the specification is increased even further by reused components (legacy code, IP). Since there is little hope that a single suitable language will replace this heterogeneous set of languages, multi-language design is becoming increasingly important for complex embedded systems.

The key problems in the context of multi-language design are the safe integration of the differently specified subsystems and the optimized implementation of the whole system. Both require the reliable validation of the system function as well as of the non-functional system properties. Current cosimulation-based approaches from both academia and industry are well suited for functional validation and debugging and yield an easy understanding of the complete system function. However, these approaches are less powerful for the validation of non-functional system properties. This is due to the known limitations of simulation such as incomplete coverage with implementation-dependent corner cases. In the context of subsystem integration, this problem is even aggravated due to the limited controllability and observability of simulation coverage.

In this dissertation, a novel compositional approach to embedded system design is presented which augments existing cosimulation-based design flows with formal analysis capabilities regarding non-functional system properties such as timing or power consumption. Starting from a truly multi-language specification, the system is transformed into an abstract internal design representation which serves as basis for system-wide analysis and optimization.

The core of the presented approach is the homogeneous internal design representation, the SPI (System Property Intervals) model. This model is based on processes communicating via unidirectional channels. The SPI model elements are characterized by a set of parameters instead of their detailed functionality. The parameters capture non-functional properties of the element such as timing, power consumption, and activation conditions. A major step towards high semantic flexibility of the model is the use of behavioral intervals for the model parameters. This allows the specification of incomplete information

and thus facilitates the integration of system parts whose internal functional details are only partially known, such as legacy code. Due to the abstraction of functionality and the behavioral intervals, SPI is a non-executable model. Rather, a SPI representation of a system bounds all possible system behaviors with respect to its non-functional properties. While behavioral intervals allow the abstraction and clustering of different process execution behaviors, process modes support their explicit, distinct specification. Using both concepts, the degree of abstraction in a SPI representation can be controlled, allowing to effectively cope with system complexity.

A fundamental principle of the presented approach is to integrate existing design environments and tools wherever possible. In this context, a crucial point for the applicability of the approach is the availability of well-defined interfaces to existing design methods and the availability of transformations from established specification languages to the SPI model in particular. Thus, in order to demonstrate the applicability, examples for model transformations (Simulink, C processes, state-based models etc.) and for the application of optimization and analysis methods with a focus on static and dynamic scheduling are given.

# Kurzfassung

Ein wesentlicher Trend im Entwurf eingebetteter Systeme ist die steigende Komplexität der zu entwerfenden Systeme. Angetrieben durch die stetig wachsende Integrationsdichte von Halbleiterbausteinen, kann eine immer größere Funktionalität zum gleichen Preis implementiert werden. Neben der steigenden Problemgröße hat die zunehmende Komplexität eine weitere Dimension: die resultierende Heterogenität bezüglich der verschiedenen Funktionen und Komponenten eines eingebetteten Systems. Dies bedeutet, daß Funktionen aus verschiedenen Anwendungsbereichen in einem einzelnen System eng miteinander kooperieren. Zum Beispiel umfaßt ein Mobiltelefon reaktive Funktionen (Protokollverarbeitung), transformative Funktionen (Signalverarbeitung) und interaktive Funktionen (Benutzerschnittstelle). Es ist in der industriellen Praxis etabliert, daß in jedem Anwendungsbereich spezialisierte Spezifikationssprachen und Entwurfsumgebungen zum Einsatz kommen. Die daraus resultierende Heterogenität der Spezifikation wird durch wiederverwendete Komponenten (Legacy Code, IP) noch verstärkt. Da wenig Hoffnung besteht, daß eine einzige geeignete Sprache diesen heterogenen Mix von Sprachen ersetzen wird, gewinnt der mehrsprachige Entwurf für komplexe eingebettete Systeme an Bedeutung.

Die Hauptprobleme im Bereich des mehrsprachigen Entwurfs sind die sichere Integration der verschieden spezifizierten Teilsysteme und die optimierte Implementierung des gesamten Systems. Beide Probleme verlangen eine zuverlässige Validierung der Systemfunktion sowie der nichtfunktionalen Systemeigenschaften. Heutige cosimulationsbasierte Ansätze aus Forschung und Industrie sind gut geeignet für die funktionale Validierung und Fehlersuche und bieten ein gutes Verständnis der gesamten Systemfunktion. Allerdings haben diese Ansätze Schwächen bei der Validierung nichtfunktionaler Systemeigenschaften. Dies liegt an den bekannten Beschränkungen der Simulation wie zum Beispiel der unvollständigen Abdeckung von implementierungsabhängigen Randfällen. Im Kontext der Teilsystemintegration wird dieses Problem aufgrund der begrenzten Steuer- und Beobachtbarkeit der Simulationsabdeckung noch verschärft.

In der vorliegenden Arbeit wird ein neuartiger kompositionaler Ansatz für den Entwurf eingebetteter Systeme vorgestellt, der existierende cosimulationsbasierte Entwurfsflüsse um Fähigkeiten zur Analyse nichtfunktionaler Systemeigenschaften ergänzt. Ausgehend von einer mehrsprachigen Spezifikation, wird das System in eine abstrakte homogene interne Darstellung transformiert, die als Grundlage für die systemweite Analyse und Optimierung dient.

Der Kern des vorgestellten Ansatzes ist die homogene interne Darstellung, das SPI-Modell (System Property Intervals). Dieses Modell basiert auf Prozessen, die über unidirektionale Kanäle miteinander kommunizieren. Die SPI-Modellelemente werden nicht durch ihre Funktionalität sondern durch eine Menge von Parametern beschrieben. Die Parameter beschreiben die nichtfunktionalen Eigenschaften des Element wie Zeitverhal-

ten, Verlustleistungsaufnahme und Aktivierungsbedingungen. Ein wesentlicher Schritt in Richtung hoher semantischer Flexibilität des Modells ist die Verwendung von Verhaltensintervallen für die Modellparameter. Dies erlaubt die Spezifikation unvollständiger Informationen und ermöglicht so die Integration von Teilsystemen, deren interne funktionale Einzelheiten nur teilweise bekannt sind (z.B. Legacy Code). Aufgrund der Verhaltensintervalle und der Abstraktion der Funktionalität ist SPI nicht ausführbar sondern begrenzt alle möglichen Systemverhalten bezüglich ihrer nichtfunktionalen Eigenschaften. Während Verhaltensintervalle die Abstraktion und Zusammenfassung verschiedener Prozeßausführungsverhalten erlauben, unterstützen Prozeßmodi ihre explizite getrennte Spezifikation. Mithilfe beider Konzepte (Intervalle und Modi) kann das Abstraktionsniveau eines SPI-Netzwerks gesteuert werden, um so effektiv die Systemkomplexität zu beherrschen.

Ein grundlegendes Prinzip des vorgestellten Ansatzes ist die Integration existierender Entwurfsumgebungen und Werkzeuge wo immer möglich. In diesem Zusammenhang ist ein wichtiger Punkt für die Anwendbarkeit des Ansatzes die Verfügbarkeit von definierten Schnittstellen zu existierenden Entwurfsmethoden und die Verfügbarkeit von Transformationen von etablierten Spezifikationssprachen ins SPI-Modell im Besonderen. Um die Anwendbarkeit zu demonstrieren, werden in der Arbeit Beispiele für Sprachtransformationen (Simulink, C-Prozesse, zustandsbasierte Modelle usw.) und für die Anwendung von Optimierungs- und Analysemethoden unter besonderer Berücksichtigung von statischem und dynamischen Scheduling gegeben.

I would like to express my sincere gratitude to my advisor, Prof. Dr. Rolf Ernst. His unique combination of sharp insight, vision and energy has made my years at IDA a both challenging and fun experience. There was hardly a discussion with him which did not lead to novel and valuable ideas. Furthermore, I would like to thank Prof. Dr. Lothar Thiele for contributing to this research from day one and for agreeing to be the co-examiner. My thanks also goes to Prof. Dr. Manfred Schimmler for chairing the examination committee and helping me to improve my volleyball.

I have had the pleasure to work on the SPI project together with some very talented people. Without them, this work would not be what it is now. To my friends and colleagues at IDA, DATE, and TIK: Thanks! Especially, I would like to thank Kai Richter, my office-mate and fellow ”specification brother”, who started SPI as my student and became the best co-worker I can imagine.

Most of all, thanks to my wife, Imke, for her endless support and love during the many weekends and after-hours of writing up this work. This thesis is as much her work as mine. And last but not least, thanks to my parents for their support during my studies and for providing a young father with a quiet and productive working environment whenever he needed it.

# Contents

# Chapter 1

# Introduction

Over the last thirty years, electronic systems ranging from personal computers over mobile phones to smartcards have more and more become an indispensable part of the daily life. Electronic systems can be classified into general-purpose and application-specific systems. General-purpose systems are designed for a variety of applications. Examples are personal computers which are used to perform such different tasks as office applications and 3D video games. Here, the system program determines the application. In contrast, application-specific systems are designed for a certain application and throughout their lifetime perform only this application. These systems typically are an integrated part of a larger system and are thus called *embedded systems*.

A possible definition for embedded systems is to classify a system as embedded if it is ”a computer system which is embedded in a technical system which does not appear as a computer system itself”. Following from this definition is that an embedded system is a computer and thus usually a digital system which interacts with a technical non-computer host system being typically analog or mechanical. Due to this interaction across domain boundaries, an embedded system typically contains sensors and actuators. While sensors provide information on the state of the host system, actuators modify the state of the host system. Thus, an embedded system typically consists of an analog as well as a digital part.

The partitioning of the embedded system functionality between analog and digital parts can be chosen according to desired system properties such as accuracy and speed. However, this work is restricted to the design of the digital part of an embedded system, i. e. an a priori system partition into analog and digital parts is assumed. Then, the digital part is characterized by having signals with discrete time and value at its primary inputs and outputs. This restriction is motivated by the fact that analog and digital design methods are fundamentally different. However, the results of the design of the digital part such as timing properties can be fed back as parameters (e. g., dead time of digital block) for a higher-level analog/digital codesign approach (e. g., [45]) which optimizes the partition into analog and digital system parts.

Thus, from the design perspective of the digital part of an embedded system, the system environment consists of the host system as well as the analog part of the embedded system. This system environment imposes constraints on the system. There are functional constraints, specifying *what* the system has to do, and non-functional constraints, specifying *how and when* the system has to do it.

The known application of an embedded system resulting from the functional constraints provides the possibility to specialize the system implementation in order to optimize system properties such as timing, power consumption or monetary cost. This optimization, however, is not just possible but typically required in order to satisfy the non-functional constraints. Furthermore, analysis is used to check the satisfaction of the constraints. Both tasks, optimization and analysis, are important parts of all embedded system design flows and are usually supported by electronic design automation tools.

## 1.1 Motivation

When reviewing current trends for embedded systems, the need of powerful electronic design automation tools to support the design becomes evident. The most important trend is the growing complexity of embedded systems. Fueled by increasing device integration capabilities of semiconductor technology, more functions can be implemented at the same cost. Based on examples from the automotive industry, the additional functions include new functions (e. g., electronic stability program (ESP), BrakeAssist) as well as functions moving from the mechanic domain to electronics (e. g., X-by-wire) due to various reasons ranging from weight and cost reduction to increased reliability. Another dimension of the growing complexity is the resulting heterogeneity with respect to the different functions and components of an embedded system. For example, a mobile phone combines functions with fundamentally different characteristics such as the user interface reacting to sporadic events (key presses) and speech processing operating on a continuous stream of data.

Another trend is the increasing importance of efficiency requirements. While cost efficiency has always been critical due to the typically high volume of embedded systems, energy efficiency is gaining importance. Staying with the mobile phone as an example, for a given feature set not only cost has to be minimized but also standby-time has to be maximized fostering power-aware implementation methods.

Together with the decreasing product lifetime requiring even shorter times-to-market, this calls for an enormous increase in design productivity in order to keep up with the technological advances. Traditionally, the way to gain design productivity is to raise the level of abstraction at which a designer models the system and let design automation tools synthesize the system according to design decisions taken by the designer. The rationale behind this increase in abstraction levels is that design decisions at high abstraction levels have more impact on the overall system performance and cost than at the lower levels. From place-and-route tools over logic and behavioral synthesis, this has led to system-level design. A side effect of this trend to higher levels of abstraction is the necessity to analyze and check the satisfaction of constraints on lower levels of abstraction (e. g., timing closure for deep submicron designs).

A prerequisite for the utilization of design automation tools is a formal and machine-readable specification. While specification at lower levels of abstraction is very much dominated by the physical and technical facts (’a gate is a gate’), at higher abstraction levels there is more freedom to characterize the design intent. This freedom is used to define specialized languages focusing on certain design aspects. Thus, there is a variety of abstract specification languages with fundamental differences in their underlying models

of computation at the system-level. Each of these languages is particularly strong or established in a certain application domain.

As mentioned above, complex embedded systems contain tightly coupled subsystems with functions from different application domains which are typically described using several domain-specific languages. For example, a mobile phone combines reactive functions like protocol processing, transformative functions like speech processing, and interactive functions like the user interface. While reactive and interactive functions are captured by state machines in a natural way, transformative functions are best described using dataflow networks. This leads to a heterogeneous multi-language system specification. Adding to the heterogeneity of the specification is the integration of reused components such as IP (intellectual property) blocks and legacy code.

The heterogeneity of embedded system specifications is matched by embedded system architectures and implementation methods. Systems-on-a-chip (SOC) integrate programmable micro-controllers and digital signal processors with specialized memories and dedicated hardware blocks communicating via specialized interconnect. Additionally, each architectural component may have a different resource sharing strategy being, e. g., static or dynamic, preemptive or non-preemptive, and priority or time driven. Similarly to the specification languages, each of the components and methods is chosen due to its suitability for the implementation of a certain function or set of functions. For example, digital signal processors (DSPs) contain a specialized loop control (zero overhead loop) in order to support fast execution of algorithms featuring nested loops which are common for multimedia applications.

The key problems in the context of this heterogeneity are the safe integration of the differently specified subsystems and the optimized implementation of the whole system. Both require the reliable validation of the system function as well as of the non-functional system properties. Current cosimulation-based approaches from both academia (e. g., [93]) and industry (e. g., [38]) are well suited for functional validation and debugging and yield an easy understanding of the system function. However, these approaches are less powerful for the validation of non-functional constraints with respect to system properties such as timing and power consumption. This is due to the known limitations of simulation such as incomplete coverage and identification of implementation-dependent corner cases. In the context of subsystem integration, this problem is even aggravated due to the limited controllability and observability of simulation coverage. Thus, the required knowledge of subsystem interaction is often not available to the system integrator [81].

While there are formal approaches to accurately and conservatively determine non-functional properties for single processes (e. g., [95]) or for single application domains or implementation methods, the use of formal methods for the complete system is inhibited by the lack of coherency between specification languages and between implementation methods.

## 1.2 Objectives

The goal of this work is to enable system-wide analysis of non-functional system properties in order to allow safe and reliable integration and optimized implementation of heterogeneously specified multi-language embedded systems.

The resulting design methodology shall facilitate the reuse of existing design environments including language-specific modeling and code generation tools as well as cosimulation and interface synthesis tools. Thus, rather than creating a completely new design environment, existing design environments shall be augmented with new techniques and a methodology to facilitate formal system-wide analysis and optimization.

Furthermore, the heterogeneous embedded system reality needs to be accounted for, i. e. the integration of IP blocks, legacy code, and third-party software as well as incompletely specified subsystems into the design flow has to be supported.

The key to achieve these objectives is to find a suitable abstraction to cope with complexity and heterogeneity inherent in the specifications and implementations of embedded systems. This abstraction should effectively use non-determinism in order to allow the system representation at variable levels of accuracy. Note that the abstract system representation is not intended to be a novel 'super language' facilitating the direct specification. Rather, it shall serve as an intermediate compositional representation capturing non-functional properties being extracted from a heterogeneous specification.

The focus of this work is on the abstract representation of heterogeneously specified embedded systems being the enabling technology for system-wide analysis and on the corresponding design methodology. While simple analysis methods are presented in order to demonstrate the validity of the approach, a formal method to the analysis of heterogeneous implementations itself is not covered in this work. The interested reader can find information on heterogeneous analysis methods in [84] and [91].

## 1.3 Outline

After a classification of the different application domains of embedded systems, several specification languages and their underlying models of computation are compared and their suitability for certain application domains is discussed in Chapter 2. Based on this presentation, the advantages of homogeneous and heterogeneous specification are discussed. Starting with a brief introduction to system-level design of embedded systems, existing multi-language design methodologies are classified and compared in Chapter 3. Together with Chapter 2, this chapter presents related work and the current state of the art.

In Chapter 4, the core of the proposed design approach, the abstract system model for heterogeneous embedded systems, is presented. Starting with the modeling concept and the model structure, the model is subsequently refined by additional concepts. Each concept is motivated and explained using examples. Afterwards, the corresponding design methodology is presented in Chapter 5. After an overview, the different design steps and the possibilities to interface to existing tools are described in detail.

Chapter 6 provides application examples to show the applicability of the proposed approach. In particular, these are modeling examples showing how various design concepts can be represented by the system model, examples on how properties can be extracted from specification languages, as well as application examples of certain analysis methods.

After a summary and outlook on open problems in Chapter 7, the appendix shows the document type definition of an XML notation representing the abstract system representation.

# Chapter 2

# Embedded System Modeling

In this chapter, different models and languages and their applications in embedded system design are presented and compared. After a classification of embedded systems into application domains, an overview of different modeling goals and modeling concepts is given. Based on these discussions, the existence of a variety of different languages and models is justified. After motivating the heterogeneous nature of complex embedded systems, a case for heterogeneous specification of these systems is presented.

## 2.1 System Classifications

Embedded system applications can be classified into different domains. These so called *application domains* are distinguished based on certain characteristics of the applications or based on the industry or field of application. While the latter criterion is evident (e. g., telecom or automotive), this section focuses on two possibilities to classify embedded systems according to their application characteristics.

The first classification criterion is the *external behavior* of the embedded system i. e. the way the system interacts with the system environment. A system performing computations on regular, typically periodic data streams (seemingly infinite sequences of data) is called *transformative*, as it mainly transforms input streams into output streams. Examples for transformative systems are digital signal processing applications such as filters, multimedia applications, and cryptographic applications. If the input data arrives rather irregularly or sporadically and the system performs computation in reaction to these input events (data occurrences of short life time), the system is called *reactive*. Protocol processing and automotive control units (e g., airbag release) are examples for reactive systems. A special case of reactive systems are *interactive* systems where the environment is a human. A typical example for interactive systems is a user interface.

The second classification criterion is the *internal organization* of an embedded system. If the primary task of a system is to perform extensive data manipulation, the system is said to be *data-dominated*. A system which has a complex state-based control flow is called *control-dominated*.

While typically transformative systems are also data-dominated and reactive systems are control-dominated, there are, e. g., data-dominated reactive systems. A typical example for such a system is a packet switch. While its external behavior is highly reactive (distributing incoming packets to different nets or clients), its internal organization is dom-

inated by data storage optimization (reducing the number of packets to be stored) whereas the realization of the actual switching functionality is simple.

Todays complex embedded systems are typically heterogeneous in terms of containing functions from different domains. For example, a mobile phone includes reactive functions (protocol processing), transformative functions (speech processing) and interactive functions (user interface).

## 2.2 Design Representations

The starting point of a systematic design flow is the formal representation of the desired system functionality. This representation can be done at different levels of abstraction and/or with a focus on certain design aspects such as timing or functionality. Such a formal representation is called a *model of computation (MOC)* or, short, model.

There is a wide variety of models of computation with differences in expressiveness (extent of representable functionality), analyzability (extent of verifiable properties), and commercial availability (tool and library support). However, there is no single best model of computation as the suitability of a model varies with the application domain (e. g., transformative or reactive) and the modeling goal (e. g., simulation or synthesis). Choosing an inappropriate model of computation may lead to a serious degradation of the design quality.

Models of computation can be seen as a set of 'laws of physics' which govern the interaction of elements of the model [22]. These 'laws' concern abstract properties such as concurrency, time, activation, or communication. Models of computation are typically implemented by a language. Then, the expressiveness or semantics of a language is determined by its *underlying* model of computation while its syntax effects the compactness, readability, and usability. Under this definition, object orientation of programming languages like C++ or Java is a syntactical concept to support systematic reuse and not a model of computation [66].

The intended use of a model or language in the context of a design methodology determines the requirements imposed on the particular model or language. A model for specification is used to capture the initial design intent. Depending on the intended use of the specification there are different flavors of specification models. One the one hand, a specification model targeted at synthesis typically contains non-determinism with respect to behavioral aspects such as timing or execution order of computational blocks. This non-determinism provides design alternatives and thus freedom which can be used in order to optimize the system implementation. On the other hand, a model targeted at simulation typically imposes a total order on computation and makes assumptions in order to simplify simulation and functional validation. In these cases, the specified system behavior is often not implementable (e. g., zero latency computation). Yet, there are standard interpretations (e. g., computation latency "sufficiently" short compared to external system timing) that allow an implementation of the specification. In summary, a specification model or language bounds the set of possible correct system behaviors.

In contrast to a specification, the implementation is usually deterministic. The non-determinism of the specification is gradually reduced by taking implementation decisions to a point where there is only a single system behavior. Note that here the term single

behavior does not mean that there is just a single path of execution but rather that given a certain input stimulus at a certain system state the system shows a deterministic response. Thus, the primary requirement for a language used for implementation is the ability to control implementation details.

Although the later presented system-level design flow in line with other modern design flows advocates the use of abstract specification models and languages, implementation languages still have to be regarded by a system-level design flow for the following reasons:

- During the design, implementation language representations of system parts may be generated by automatic code generators. Thus, implementation languages essentially form an intermediate design representation.

- Host code that adheres to an abstract coordination model (e. g., dataflow process networks) is often hand-written in an implementation language.

- As systems are rarely designed from scratch, part of the system functionality consists of components from previous designs (software: legacy code). These components are typically not available in abstract specification languages but rather in implementation languages. Similarly, soft IP is usually provided in a hardware description language.

- And last but not least, a significant portion of the system functionality is still being developed directly in implementation languages, e. g., due to high start-up costs of specification tool environments, tradition, or the need to directly control implementation details in order to achieve a high design quality.

In this section, common models of computation and the languages implementing them are introduced. Based on the level of abstraction and their intended use, models and languages for specification are distinguished from models and languages for implementation.

Before comparing the different models and languages, the concept of process networks along with some general terminology common to many models of computation is introduced.

### 2.2.1 Process Networks

Many models of computation are based on the notion of *process networks*. Process networks provide an intuitive separation of computation and communication with network nodes being processes connected by directed edges which represent communication links. The semantics of a process network are defined by a *host model* providing the computation semantics and a *coordination model* providing the interaction semantics. In this context, the interaction includes process communication as well as process activation.

In order to allow an abstract view at process communication, the communicated data is typically abstracted to indivisible quantums of data, called *tokens*. The information a token contains can be described at different levels of precision. The VSIA model taxonomy [25] distinguishes five levels of data precision (token, property, value, format, bit logical). While the token precision level does not specify the content information at all, the other four levels specify the content information with a gradually increasing level of

implementation details ranging from abstract properties such as colors to exact bit representations of these properties. Then, processes communication is represented in terms of writing and reading tokens and can be classified based on the following properties:

- **destructive or non-destructive.** A write operation is destructive if it overwrites previous tokens and non-destructive if previously written tokens are unaffected by it. A destructive read operation can access a certain token only once (as it consumes or removes that token) whereas a non-destructive read can access the same token many times.

- **blocking or non-blocking.** A write operation is blocking if it cannot be completed if its completion would result in the loss of current or previously written tokens, e. g., due to a buffer overflow. Similarly, a blocking read operation cannot be completed if it has to wait for some tokens to be written. In contrast, non-blocking communication constructs can be completed independently of the state of the process network.

- **synchronous or asynchronous.** Synchronous communication assumes that corresponding read and write operations are executed at the same time, whereas asynchronous communication assumes buffers where the written data is stored so that it can be read later.

- **unidirectional or bidirectional.** Unidirectional communication links support only a single communication direction while bidirectional communication links allow the exchange of tokens in both directions.

- **point-to-point, multicast, or broadcast.** These properties differ in the number of recipients of a written token being one (point-to-point), many (multicast), or all (broadcast) other processes in the network. Obviously, with edges in a network having exactly two connected processes, only point-to-point communication can be explicitly represented in process networks. Although multicast and broadcast can be represented explicitly using several communication links they are often implicitly specified in the host language (e. g., processing graph method (PGM) [65]).

The life time of a token starts with its generation (production) and ends either by being consumed (destructive read) or by being overwritten (destructive write). However, there is also the possibility that tokens exist only at certain discrete points in time and are implicitly destroyed by an advancement of time. Tokens with such a limited life time are typically called *events*[1]. Yet, there are models of computation (e. g., CFSM [15]) specifying event buffers effectively abolishing the destruction by time advancement.

Process activation determines under which conditions a process may execute. While the activation of processes can be arbitrarily complex, there are two common activation principles for process networks. An OR-activated process is activated and thus ready to start if an input condition is satisfied on one of its incoming communication links. In contrast, an AND-activated process is only activated if the conditions on all of its

[1]This is different from the use of the term event in the context of discrete-event systems where an event denotes the change of a continuously present signal or variable.

incoming communication links are satisfied. However, process activation may also be decoupled from token communication and, e. g., depend on time.

The partition of the system in computation and communication is a natural separation of concerns which facilitates and supports reuse. Furthermore, many properties of process networks can be determined based on the coordination model without considering the host model. This is the reason why many modeling approaches allow 'arbitrary' host models. However, this total neglection of the host model carries a risk which is a typical source of design errors. This is the possible inconsistency of host and coordination model with respect to communication. If, e. g., the coordination model is solely based on point-to-point communication but the host model allows and employs implicit broadcast, an analysis method based on the coordination model may yield incorrect results due to the 'hidden' broadcast [69].

Possible further features of process networks are hierarchical extension of nodes and dynamic instantiation of nodes and networks. Hierarchy is a suitable structuring means, e. g., to reduce the conceptual design complexity by increasing readability and thus to increase the comprehension of the system functionality and modeling efficiency. Dynamic instantiation of nodes and networks greatly enhances flexibility of a process network. However, even if supported by a design representation, dynamic instantiation is often not (yet) supported by its corresponding embedded system design tools due to strict limitations imposed on an embedded system such as, e. g., bounded memory and due to difficulties to reliably analyze the behavior of systems relying on dynamic instantiation.

Furthermore, process networks may show non-deterministic behavior, i. e. the system behavior given the process network representation and a certain set of stimuli (input data) can not be exactly predicted. Non-determinism in a process network can have two different sources and benefits. One possible benefit is that the choice of different correct system behaviors (e. g., process execution ordering or several functional alternatives) is not unnecessarily limited at the specification stage. Rather a non-determinate specification leaves the choice of system behavior to the system optimization step. The other possible source of non-determinism is abstraction of influences and details in order to obtain a more compact representation.

All models of computation presented in this chapter can be seen as specializations of process networks which define certain host and coordination models in order to yield desirable model properties.

### 2.2.2 Specification

In the following, common design representations used for the specification of the system function are introduced. The focus is on models of computation rather than on languages since from a modeling point of view the underlying model of computation of a language is far more important than its syntax. For illustration, language examples for the presented models are given as well.

A commonality of the presented models is that they capture the desired system function while abstracting implementation details. Thus, the system function is homogeneously specified without distinguishing between functional blocks intended for hardware or software implementation. This allows for specification reuse and provides maximum freedom for design space exploration.

The presented models are classified as being *activity-based* or *state-based* [30]. An activity-based model describes a system in terms of activities or functions and their data and execution dependencies while a state-based model represents a system as a set of states and transitions between these states. The suitability of an activity- or state-based model for the description of a certain system function depends on the application domain of the function and will be discussed in the respective sections. Afterwards Petri Nets, a model of computation which defies this classification as it is capable to represent both modeling styles, is presented.

Another possible classification criterion of models which is orthogonal to the activity-/state-based classification is the representation of timing. Here, two possible classes are *abstract or explicit timing*. An example for abstract timing is the *synchronous* abstraction denoting time as a sequence of steps with indefinite spacing where computation and communication are performed in zero time at these steps. In contrast to this, the *asynchronous* abstraction defines model execution by a partial ordering on computation and communication which both may take an indefinite time. An example for explicit timing is the specification of exact times for characteristic points of a model execution such as output data production in terms of discrete time steps.

#### 2.2.2.1 Activity-Based Models

The most general form of an activity-based model is a dataflow graph where nodes represent functions and edges their data dependencies. In the following, different activity-based models of communication are presented which put restrictions on the functions and dependencies in order to yield desirable model properties.

**Kahn Process Networks** Kahn process networks [58] are a special class of process networks where processes operate on infinite sequences of tokens, so called *streams*. Each token is written and read exactly once. This is accounted for by the coordination model specifying communication links to be unidirectional, unbounded FIFO queues. These queues are accessed by processes through non-blocking, non-destructive write, i. e. processes immediately add tokens to queues, and blocking, destructive read, i. e. a process consumes (removes) input tokens from queues and stalls if the accessed queue does not contain at least the number of tokens required by the read. Each queue has exactly one reading and one writing process.

Furthermore, Kahn constrains processes to be sequential and continuous mappings of input streams to output streams. From allowing processes to map streams instead of tokens follows that processes may have and manipulate state. The *continuity* property can also be formulated as

$$f(x_1 x_2 \ldots x_n) = f(x_1) f(x_2) \ldots f(x_n)$$

with $f(X)$ being the output stream generated by the process based on an input stream $X = x_1 x_2 \ldots x_n$. Kahn further shows that continuous processes are also monotonic where *monotonicity* is defined as

$$X \subseteq Y \Rightarrow f(X) \subseteq f(Y) \quad \text{or} \quad f(x_1 x_2 \ldots x_n) \subseteq f(x_1 x_2 \ldots x_n x_{n+1})$$

where $X \subseteq Y$ denotes the containment of stream X in stream Y, i. e. stream Y extends stream X by additional elements.

Both properties are extremely useful as a key consequence of these properties is that processes can be executed iteratively i. e. a process does not need to have all its inputs, i. e. the complete possibly infinite input streams, to start execution. Furthermore, it will not wait forever (on an infinite amount of input data) until it produces output data. From monotonicity in this context follows that additional input data can only trigger the process to produce additional output data but never to invalidate already produced output data.

Based on these findings, Panagaden et al. show in [78] that networks of monotonic processes with an arbitrary network structure are themselves monotonic. Assuming defined initial process states, Lee shows that monotonic Kahn process networks are determinate in the sense that internal and output streams depend only on input streams [69]. From this follows that a function described as a Kahn process network is independent on the order of process executions as the blocking read semantics naturally accounts for the partial execution order as defined by the data dependencies. This property is excellent for design space exploration as it provides maximum freedom to optimize the process execution order, e. g., to minimize system response times or memory size. Furthermore, Kahn argues that a continuous network has a minimum fixed-point i. e. a minimum cycle or sequence of process executions that transfer the network back to its initial state. This guarantees a bounded schedule length.

Kahn presents in [58] a minimalistic language implementing Kahn process networks. A current language example is YAPI [23], a C++ class library providing an application programming interface for capturing functional specifications of digital signal processing applications.

**Dataflow Process Networks** Dataflow process networks [69] are a special case of Kahn process networks where processes are decomposed into sequences of indivisible quantums of computation called actor firings. During each firing an actor consumes input tokens and produces output tokens. When an actor may fire is defined by firing rules specifying the type and number of data tokens which have to be available at the actor inputs in order to allow an actor to fire. This decomposition into firings helps to reduce the context switching overhead inherent in many direct implementations (e. g., demand-driven [59]) of Kahn process networks [92].

A dataflow process network where actors have a fixed production and consumption behavior, i. e. actors always consume and produce certain fixed numbers of tokens, is called a *synchronous dataflow (SDF)* process network [68]. The firing sequence or execution order of an SDF process network can be determined statically thus allowing a very efficient implementation [67].

In the *cyclo-static dataflow (CSDF)* model [5], the number of consumed and produced tokens by an actor may vary cyclically, e. g., an actor may consume one token at its odd firings and consume three tokens at its even firings. This still yields a statically determinable firing sequence as for an CSDF network a corresponding SDF network can be created by unfolding [5].

The more general *dynamic dataflow (DDF)* process networks allow actors to have an input-data dependent token production and consumption behavior. An example is the *Boolean dataflow (BDF)* model [8] where the numbers of consumed and produced data

tokens depends on the value of a token read from a dedicated control input. While this results in Turing completeness, this also causes a BDF process network to be in general not statically schedulable.

Besides these approaches, there are several other possibilities to define coordination models of dataflow representations including the computation graphs of Karp and Miller [60] and the processing graph method (PGM) [65].

Dataflow process networks are well-suited for the transformative domain as the FIFO buffers and the only partially specified process execution order naturally support the optimization of system properties such as throughput or memory size. However, they are ill-suited for the reactive domain as the FIFO buffers inhibit short reaction times to incoming events, e. g., a token signaling an emergency halt can not bypass previously written tokens and has to wait in a queue until these tokens are consumed and processed.

The above discussion of dataflow process networks covered only the coordination semantics as they determine the interesting model properties. In the range of the allowed process behavior as specified by the coordination semantics, the function performed by a dataflow process can be given in various ways depending on the chosen design environment. Different possibilities are the description of a process by another dataflow process network (hierarchical extension), predefined functional library blocks (e. g., COSSAP [90] from Synopsys), or implementation languages such as C (e. g., COSSAP) or Java (e. g., DF domain of Ptolemy II [22]). Other design environments based on dataflow process networks are SPW [10] from Cadence and DSP Station [37] from Mentor Graphics.

**Simulink** After having presented theoretically clean models of computation above, in the following a specification language is presented which is established industrial practice. Its underlying model of computation combines aspects of different clean models of computation such as synchronous/reactive (discrete time steps) and discrete time [22] (strict periodicity).

Simulink [74] from TheMathworks is a block diagram based commercial tool and language for the modeling and simulation of continuous-time and discrete-time systems. There is wide variety of supporting tools (e. g., automatic C code generators) and libraries with predefined functional blocks. A typical application of Simulink is the joint modeling of a physical system (continuous-time) and its associated control and signal processing functionality (discrete-time). While the model of the physical system is eventually replaced by the actual system, the control and signal processing functionality is typically implemented as digital embedded system. Thus, in the context of this work, the focus is restricted to Simulink's discrete-time domain.

While being also an activity-based model, Simulink's underlying model of computation is very different from Kahn or dataflow process networks. In Simulink, processes are called blocks and communicate via unidirectional edges representing signals. From a token-based communication point of view, these signals have register semantics, i. e. blocks access signals via destructive write and non-destructive read.

Simulink has an idealized computation and communication timing. All activities happen infinitely fast at exact points in time. For each block, a fixed output delay can be specified in terms of a predefined number of time steps. Blocks have associated sample times selecting the time steps in which they are executed i. e. they execute exactly pe-

riodically. An exception are triggered blocks which are executed when their associated triggering condition, a function on the value of a signal connected to a dedicated trigger port, is satisfied. In each time step the block driving the trigger input is executed, the triggering condition is evaluated leading to a possible execution of the triggered block in the same time step.

Within a time step, the signals impose a causality i. e. the block writing a signal is executed before a block reading the signal if both blocks are executed at the same time step. Obviously, this leads to a conflict for a cycle in the network structure without delay. Thus in Simulink, cycles have to incorporate at least one block with an associated output delay.

The complexity of blocks varies from simple add or multiplication operations to complex filter algorithms. The function of a block can be either specified in MATLAB or C or taken from a library containing predefined functional blocks for common functions. Furthermore, hierarchical extension of blocks is possible.

The Simulink model of computation is well-suited for simulation. However, its strict timing model is too restrictive for efficient implementation. This becomes evident when looking at the problems of automatic code generators for Simulink [57]. Simulink is targeted at the transformative domain and is particularly suited for control engineering and digital signal processing. In comparison to dataflow process networks, reactivity is better supported due to the concept of triggered blocks.

#### 2.2.2.2 State-Based Models

The basic form of a state-based model is a finite state machine consisting of states and transitions. But in order to efficiently describe complex systems, different compositions of finite state machines are used.

**Finite State Machines** A traditional finite state machine (FSM) (e. g., [66]) is defined by:

- a set of input events,
- a set of output events,
- a finite set of states with a distinguished initial state,
- an output function computing output events based on current state and input events (assuming Mealy semantics), and
- a next state function (or transition function) determining the next state based on the current state and input events.

Traditional finite state machines are a synchronous model, i. e. the reaction to input events is instantaneous. Thus, evaluation of the output and next state functions happens infinitely fast. While FSMs are typically used as host model of process networks, there is also an interpretation of FSMs where states are represented as processes and state transitions as well as input and output events form synchronous non-blocking communication links [46]. FSMs have been extended in various ways to allow the manipulation of data variables. An overview can be found in [46].

Traditional FSMs are well-suited for modeling sequential systems. Representing concurrency or large amounts of data, however, is problematic as the required state space 'explodes', i. e. the number of global states grows exponentially with the number of states of concurrent functions or with the number of values of each data object in the system. A solution to the state explosion problem is to compose traditional finite state machines to explicitly represent concurrency. This reduces the size of FSMs and thus their visual although not computational complexity. Based on the used coordination model, synchronous and asynchronous FSM composition is distinguished.

**Synchronous Composition of FSMs** A process network with processes being represented as FSMs (possibly trivial i. e. single state) and a synchronous non-blocking broadcast communication between the FSMs forms a model of computation called *synchronous/reactive* [40]. The synchronous/reactive model assumes a step-wise system execution where all processes simultaneously read their inputs, perform their computation, and produce their outputs. Clearly, this implies zero-delay computation while between time steps an indefinite time passes. The processes communicate via events which usually carry a value and are valid only at the time step in which they are produced. Output events are immediately available to all other processes via broadcast. Thus in contrast to the explicit data flow of the activity-based models, there is no explicit event flow.

Assuming no cyclic dependencies of events at a single time step, the synchronous/-reactive network has a single equivalent FSM but provides a more compact and modular representation of the system. Examples for languages implementing the synchronous/-reactive model of computation are Esterel [4], Argos [72], and Harel's StateCharts [42, 41] implementing the most well-known synchronous compositional approach for FSMs, hierarchical concurrent finite state machines (HCFSM).

The exact event timing and the instantaneous reaction define a total order on the process execution leading to an exact and complete definition of the system response for a given input pattern. This is excellent for system simulation but limits the design space. The impact of this limitation depends on the application and on the target architecture. A fast computation compared to the system environment timing, as common for reactive, control-dominated systems, matches the model of computation and does not infer implementation penalties. For data-dominated systems with tight timing requirements, however, the limitations of the complete execution order are severe. Furthermore, the distributed implementation of a synchronous/reactive system requires extensive synchronization [16, 17] and infers an additional implementation penalty for heterogeneous architectures where the speed of the slowest resource limits the speed of the other resources.

**Asynchronous Composition of FSMs** In order to remedy some of the inflexibility caused by a purely synchronous model, models of computation have been proposed that combine synchronous and asynchronous concepts in a single model. They differ in where the boundary between synchronous and asynchronous behavior is chosen.

One example model of computation which advocates a so called "locally synchronous - globally asynchronous" paradigm are the *Codesign Finite State Machines (CFSM)* [15] as used in the Polis codesign environment [2]. The differences between a CFSM and a traditional FSM are that transitions may trigger data computation and take an unbounded

but finite reaction time. CFSMs are locally synchronous in consuming their inputs simultaneously in zero-time and appear globally asynchronous as several CFSMs in a network no longer change state and produce output events simultaneously. A CFSM network consists of CFSMs communicating via one-place event buffers with destructive read (blocking) and write (non-blocking) access. As transitions can test the absence of an event, the system behavior depends on the event arrival order and thus is fundamentally implementation-dependent.

Another example is the *Specification and Description Language (SDL)* [49], a formal language for specification, simulation and design which is widely used in the telecommunication industry. From a process network perspective, SDL consists of processes representing extended FSMs and unbounded FIFO queues forming the communication links. In contrast to Kahn process networks, however, each process has just a single input queue into which all other processes may write their output tokens. Furthermore, processes have full control over their input queues including a save construct essentially allowing out-of-order consumption.

All SDL processes are clustered to blocks. These blocks are not only a structuring concept but also form the boundary between synchronous and asynchronous behavior. While processes inside a block are synchronously composed i. e. all processes simultaneously consume one event from their input queue, change state, and produce output events which are immediately available for the receiving process, composition of blocks is asynchronous in the sense that events between processes from different blocks are delayed for an unbounded but finite time. Thus, blocks are usually used to capture processing element boundaries as the implementation of synchrony on a single processing element is simple compared to enforcing synchrony over a distributed system. A discussion of additional features of SDL such as timers and dynamic process and network instantiation can be found, e. g., in [7, 44].

Another design representation based on the composition of FSMs is the *SOLAR* model [53]. The system is represented in terms of DesignUnits which are specified by extended hierarchical FSMs. The DesignUnits can be hierarchically nested and communicate via remote procedure calls encapsulated by ChannelUnits. This encapsulation completely decouples computation and communication and allows the representation of various communication schemes including all of the above mentioned composition mechanisms. This flexibility with respect to communication makes SOLAR a very suitable representation for communication refinement and synthesis.

#### 2.2.2.3 Petri Nets

A model of computation which can be assigned to neither activity- nor state-based models are *Petri nets* [79] as it is capable to represent both modeling styles. A Petri Net is a bipartite graph of transitions and places connected by directed edges. Each place has a number of tokens which are not ordered and at least in the basic Petri net model do not carry additional information. Transitions may fire if each of its input places contains at least as many tokens as there are edges from the input place to the transition. At firing, a transition removes one token via each incoming edge and produces a token via each outgoing edge.

By associating activities with transitions and interpreting places as dependencies between these activities, Petri nets can be used as an activity-based model. However, by interpreting places as states and transitions as state transitions, Petri nets can also be used as a state-based model where the current state is denoted by the place which contains a token.

Petri nets are usually not used to specify a system but rather to prove system properties like liveness, deadlocks, and boundedness. For this, systems or rather certain aspects of systems are modeled based on Petri Nets for which a wide variety of formal analysis methods exist. The basic Petri net model, however, is limited in expressiveness. Thus, there are several extensions such as associating colors with tokens in colored Petri Nets [52], associating time with transitions or tokens (time stamps) [99], or prioritizing transitions in order to define an order for concurrently activated transitions. While these extensions increase the expressiveness, they also degrade the analyzability.

In general, Petri nets are a very flexible model of computation that are suitable for analysis of reactive as well as transformative systems. Yet, it should be noted that Petri nets have been extended in so many directions that the only commonality remaining are probably the concepts of non-destructive write, destructive read and activation. In this sense, even Kahn graphs can be understood as a special class of Petri nets. While acknowledging this ”wide angle” view of Petri nets, this view is less useful when discussing specific model properties.

### 2.2.3 Implementation

Implementation languages include the traditional design languages for embedded systems. These are typically imperative programming languages such as C for system parts being implemented in software and hardware description languages such as VHDL for system parts implemented in hardware. These languages are often called high-level languages (e. g., VSIA model taxonomy [25]) but from a system-level design perspective they form a lower level design representation compared to the more abstract specification languages. Furthermore, the term implementation language more accurately reflects the most prominent feature of these languages in comparison to specification languages, the ability to control implementation details. Thus, they are typically used in a system-level design flow to represent system-level design decisions.

Recently, implementation language extensions have been proposed which are aimed at closing the semantic gap to specification languages. This is done by raising the level of abstraction in order to allow the functional description to be independent of software or hardware implementation. The proposed way to achieve this goal is to augment a software implementation language, namely C or C++, with concepts like parallelism and various communication primitives. While there have been previous approaches from academia such as $C^x$ [26] and HardwareC [63], the recent approaches find a wide industrial support in terms of tools and environments.

**SpecC** One of these approaches is the SpecC language [31] which is a superset of ANSI-C. In SpecC, the system function is described by means of communicating functional blocks called behaviors which can be composed in various ways in order to support structural as well as behavioral hierarchy such as sequential, concurrent, or pipelined execu-

tion. Furthermore, specification language features such as abstract FSM specification are supported. In order to execute a SpecC representation, it is translated into an intermediate C++ program which is then compiled using a standard compiler.

The SpecC language is accompanied by a corresponding methodology based on models on four different levels of abstraction and a stepwise refinement between these models. The models are all represented using the SpecC language but for each abstraction level a corresponding modeling style is suggested [33].

**SystemC** An approach directly based on C++ is the SystemC language [47] which is an extensible class library allowing the specification of parallelism and communication. While SystemC 1.0 is basically a hardware description language, SystemC 2.0 provides extended modeling possibilities such as client-server communication. Furthermore, SystemC claims to allow domain-specific modeling, e. g., the representation of Kahn process networks. However, this is still work in progress. SystemC is heavily supported by electronic design automation companies with tools, e. g., for the compilation of a SystemC model (adhering to a synthesizable subset) to hardware.

## 2.3 Homogeneous vs. Heterogeneous Specification

As seen from the example of a mobile phone in a previous section, complex embedded systems include reactive as well as transformative functions which are tightly coupled. The presented specification models or languages, however, are targeted at a single domain only. This gives rise to the question of how to specify such a complex embedded system. The two possible approaches are a heterogeneous multi-language specification consisting of several differently specified system parts or homogeneous specification using a single general language.

Obviously, implementation languages such as SpecC or SystemC are general enough to represent the desired functionality. However, they do not match the abstraction levels of the respective domains thus being not intuitive for the designer to use. A consequence of this is that an implementation language representation is inefficient (e. g., in terms of lines of code) compared to an abstract domain-specific representation. But an even more important disadvantage is that the domain-specific properties used for transformations and optimizations are lost within the detailed description and have to be reidentified and abstracted in order to be used during system optimization and synthesis. Examples are the communication parameters (number of produced and consumed tokens) of dataflow process networks or the freedom of process execution order in Kahn process networks. Thus, homogeneous modeling at this low level of abstraction is not suitable for system optimization and synthesis.

Yet, there are languages and design environments trying to cover mixed reactive/-transformative embedded systems at a higher level of abstraction. Examples are:

- STATEMATE [43] where the state-based StateCharts model targeted at the reactive domain is composed with ActivityCharts, a hierarchical dataflow diagram for the representation of transformative functions,

- Simulink/Stateflow [75], an extension of Simulink with blocks representing hierarchical concurrent finite state machines being evaluated with a discrete sample time, and
- CoCentric System Studio [89], allowing the parallel and hierarchical composition of FSMs and dataflow models.

While the resulting languages are different, they essentially combine different model semantics by defining an interaction model. Thus from a strict perspective, they are heterogeneous or multi-language representations.

A key advantage of heterogeneous or multi-language specification approaches is the possibility to include the traditional domain-specific design environments of the application developers in the system design flow. This allows to reuse existing domain-specific transformation and optimization techniques and tools as well as existing designs and to utilize the designers' experience with respect to tools and formalisms.

The problems of multi-language specification and design approaches such as the lack of coherency regarding languages and methods are the topic of Chapter 3.

## 2.4 Summary and Conclusion

This chapter has shown that there is a wide variety of languages and models of computation for the representation of embedded systems. This variety is justified by the fact that the expressive and analytical power of a given model of computation depends on the application domain. In other words, there is no single best language or model as each application domain has its specific languages. Considering additionally the large investments in existing design environments including the training and expertise of application developers, it seems improbable that a novel super language will replace the existing mix of languages.

Furthermore, it has been shown that complex embedded systems are heterogeneous in nature as they include tightly coupled functions from different application domains. Then, from the above observations follows that complex embedded systems are typically heterogeneously specified such that multi-language design approaches have to be considered for embedded system design.

Another lesson to be learned from this section is the role of abstraction in embedded system modeling. For dataflow process networks, it was shown that scheduling methods and design space exploration can be applied without knowledge of a process functionality but only based on abstract properties such as token production and consumption as well as timing information. Another advantage of abstraction is the ability to effectively cope with system complexity.

# Chapter 3

# Multi-Language Design Methodologies

After a coarse grain overview of system-level design steps, two different classes of approaches to multi-language design of embedded systems as well as example approaches from both classes are presented. Based on a discussion and comparison of the approaches, requirements for a novel approach to embedded system design are derived.

## 3.1 System-Level Design

In contrast to design tasks starting from lower levels of abstraction such as logic synthesis or software compilation, there is no dominant or well-established methodology for system-level design of complex embedded systems. Thus, rather than presenting a complete design methodology, this section introduces the coarse grain system-level design steps in order to provide a background for the ensuing discussion of multi-language design approaches.

Figure 3.1 gives an overview of these design steps and their dependencies. Starting from a specification, the design space is explored, and based on the taken design decisions (such as target architecture selection) the system is synthesized i. e. implemented. During the whole design flow, the design is guided and results are checked by validation. The arrows between the design steps point in both directions in order to denote the mutual dependencies between the design steps leading to a highly iterative design flow.

**Specification** Input for a system-level design flow is a formal, abstract specification which defines the set of feasible design solutions i. e. system implementations. This abstract specification typically consists of a functional specification describing the desired system functionality and non-functional constraints capturing desired bounds on system properties.

The functional specification of a complex embedded system is typically heterogeneous and contains blocks described in several different specification languages, system parts from previous designs such as legacy code, and IP blocks from other companies. In order to support functional validation, such a specification is typically executable i. e. simulatable. In the presented case of a heterogeneous specification, this is achieved using cosimulation.

The non-functional constraints can be divided in technical constraints concerning system properties such as timing, power consumption, weight, or size and non-technical

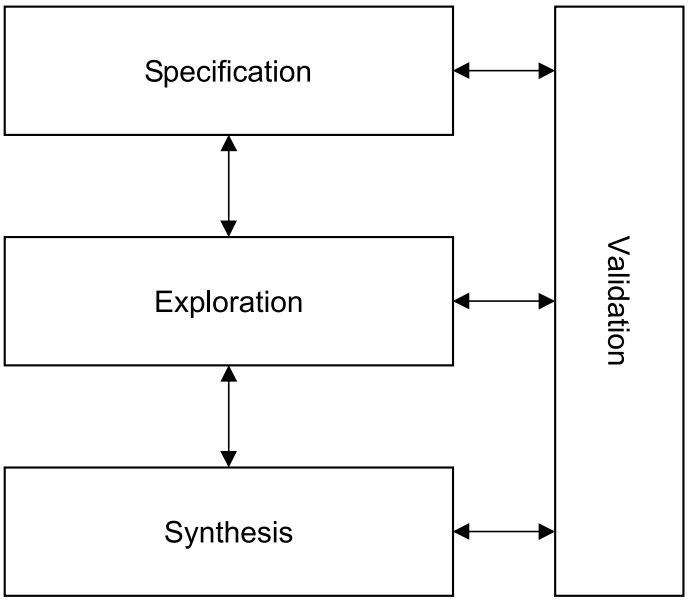



Figure 3.1: Coarse grain overview of a generic system-level design flow

constraints concerning the system cost but also the time-to-market which is rather a property of the design flow than a system property. The former constraints are imposed by the system environment requiring certain response times or a maximum heat dissipation, the latter by a marketing department having defined a maximum system cost or a certain market window for the product the system is embedded in.

While the constraints require a system optimization and specialization, the fixed functionality of an embedded system provides an excellent possibility for optimization and specialization.

**Exploration** Based on the specified set of feasible solutions, the goal of the exploration step is to find the best feasible solution for a given objective function. This clearly is an optimization problem and heavily relies on analysis and estimation methods in order to assess the different solutions. In the following, the exploration step is often called analysis and optimization step.

The solution parameters of this optimization problem depend on the abstraction level in terms of granularity and the degree of freedom left for design decisions. In comparison to, e. g., software compilation concerned with assigning operations to functional units where the decision to implement a certain function as software on a given target processor is already taken and limits the remaining design space, the granularity at the system-level (processes and components) is more coarse grain and the freedom for design decisions is larger. In practice, however, the solution space is limited by the availability of components and the used design methodology. In particular, the following design decisions are taken at the system-level exploration step:

- **Allocation** Selection of the architectural components such as processing elements (including processors as well as application-specific hardware), memories, and buses but also operating systems; determination of their respective component parameters

(e. g., for memory capacity, word length, or organization); composition of the selected architectural components (i. e. how many components are allocated and how they are connected) yielding the *target architecture*.

- **Binding or Partitioning** Mapping of the processes, communication, and data objects to the components of the selected target architecture.

- **Scheduling** Selection of the execution order of processes and communication requests (*arbitration*); determination of the system-level control and data flow.

These three tasks are highly interdependent and can not be separated in order to guarantee the determination of a global optimum. However, due to the huge design space that these tasks are spanning, usually a heuristic and iterative application of these tasks interleaved with analysis and estimation steps characterizes the design space exploration. A special subclass of system-level design flows is the platform-based design [14] where the target architecture is given as a (possibly parameterized) template and only binding and scheduling need to be performed. Platform-based design is typically supported by platform-specific tools being directly targeted to the specific design space of the platform.

The result of the design exploration is a refined specification which includes the functional specification along with the constraints, the target architecture and a complete mapping of elements of the functional specification onto elements of the target architecture.

**Synthesis** The synthesis step generates the system implementation based on the design decisions taken in the exploration step. For heterogeneous HW/SW implementations, this step is usually divided in hardware, software, and interface synthesis. While all these problems are again optimization problems with their own exploration within the remaining degrees of freedom, from the system-level perspective they provide a back end.

**Validation** Throughout the design flow, validation is used to check the correctness and impact of the performed design steps. In the specification phase, the functional correctness of the design intent is validated. During design space exploration and synthesis, validation of design decisions typically with respect to non-functional properties such as timing or power consumption is performed. This includes, e. g., the identification of bottlenecks or under-utilized resources.

The industrial state-of-the-art for validation of embedded systems is simulation which can be applied at different levels of detail. Formal approaches are becoming increasingly important but often suffer from state explosion due to system complexity.

For system-level design flows starting from multi-language specifications, there are two fundamentally different classes of approaches which will be discussed in the following sections: cosimulational and compositional approaches.

## 3.2 Cosimulational Approaches

### 3.2.1 General Concept

*Cosimulational approaches* to multi-language embedded system design assume a true multi-language specification. This means that subsystems are specified individually using domain-specific languages and their corresponding design environments. Furthermore, using these environments, the different system parts are also separately designed and validated.

The interconnection of the different system parts is usually specified in a configuration file providing a kind of system-level netlist. Based on this configuration file, the subsystems are systematically integrated using a backplane communication protocol which can be used for cosimulation as well as for the implementation using interface synthesis and communication refinement.

Throughout the design flow, cosimulation, with setups automatically generated from the configuration file, is performed on different levels of abstraction to test the system integration. The typical abstraction levels for cosimulation are:

- At the *specification level*, the inter-language communication is defined in terms of generic communication functions such as send() and receive() functions. Here, untimed cosimulation validates the system function independent of any implementation.

- At the *architectural level*, the system is partitioned and mapped onto a selected target architecture. Communication protocols are selected and communication is typically modeled at the driver-level i. e. in terms of register accesses. Using explicit synchronizations and timing estimates, cosimulation at this level validates the system partitioning as well as the selected protocols.

- At the *clock-cycle accurate level*, the interfaces are refined to wires and the communication is modeled by setting signal values. Hardware and software parts of the system are simulated at the register transfer level (RTL) and using a cycle-accurate processor simulator, respectively. The primary goal of cosimulation at this level is timing validation.

A coarse grain design flow of a generic cosimulational approach is depicted in Figure 3.2. In the following, existing cosimulational approaches classified based on their abstraction level are discussed. Here, the focus is on the design flow and its features and not on the specific implementation of the cosimulation. An instructive overview of different methods for simulator coupling is presented in [54].

### 3.2.2 HW/SW Cosimulation

Most of the existing commercial cosimulation approaches are based on implementation languages and assume an already partitioned system description. Typically, the hardware part is modeled at the register transfer level using a hardware description language such as VHDL or Verilog and the software part is described using C or C++. Recently, also SystemC has been increasingly supported.

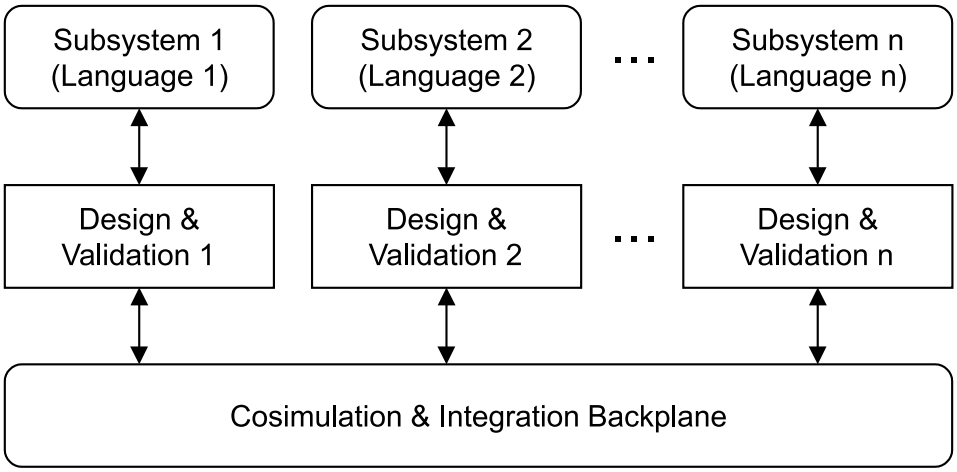



Figure 3.2: Generic design flow of cosimulational approaches

**Seamless CVE** The Seamless CVE hardware/software coverification environment [38] from Mentor Graphics is the most widely-used cosimulation tool suite. It is specifically targeted at the architectural and clock-cycle accurate levels and supports all popular embedded processors as well as various hardware simulators. The major benefit of Seamless is the ability to dynamically vary the cosimulation accuracy and speed. However, Seamless is solely targeted at cosimulation and does not provide interface synthesis or communication refinement methods. Thus, Seamless is not a complete cosimulational design approach but provides the cosimulation facilities other approaches (e. g., ArchiMate [1] from Valiosys) are based on.

**CoWare N2C** In contrast to Seamless, N2C (napkin-to-chip) [20] from CoWare integrates cosimulation with interface synthesis capabilities. N2C assumes a system specification based on C/C++ including extensions to define blocks and the ability to thread multiple blocks for concurrent execution (CoWareC). Additionally, also SystemC is supported. Starting from abstract communication protocols implemented based on remote procedure calls, a multi-layered interface synthesis refines the communication between the functional blocks by automatically generating device drivers and synthesizing hardware glue logic consistent with the selected communication protocols.

CoWare N2C allows the system to be simulated at different abstraction levels including untimed, instruction accurate, bus-cycle accurate, and clock-cycle accurate. While simulation at the first three abstraction levels is performed through executable CoWareC or SystemC models of the system, the clock-cycle accurate simulation is performed by executing the software binary code on the selected processor's instruction set simulator, and the hardware with an RTL simulator (VHDL or Verilog).

### 3.2.3 Multi-Language Cosimulation

Recently, cosimulational approaches at a higher level of abstraction as compared to the above C/HDL cosimulation have been proposed. These approaches allow generic cosim-

ulation at multiple abstraction levels using multiple languages including abstract specification languages. As the simulation speed degrades with an increasing level of detail, the incorporation of abstract specification languages allows a more efficient cosimulation earlier in the design flow (true system-level cosimulation).

**MUSIC** The MUSIC design environment [19] developed at the TIMA laboratory is a complete cosimulational approach including refinement of subsystems and communication from the specification level down to the clock-cycle accurate level. Throughout the design flow, the system and its refinement steps may be validated via cosimulation allowing the combination of subsystems being at different levels of abstraction.

The cosimulation supports a heterogeneous specification consisting of subsystems specified in C, VHDL, SDL, Matlab/Simulink, and COSSAP (DF process networks). However, the subsystem refinement is only possible for C, VHDL, and SDL. The interconnections between the different subsystems are represented by a configuration file based on the SOLAR format [53]. In SOLAR, the system is modeled as several design units communicating via remote procedure calls (RPC) on different levels of abstraction (high-level channels to physical signals). Based on this configuration file MUSIC generates cosimulation setups as well as communication protocols for implementation.

**Valiosys ArchiMate and CosiMate** The ArchiMate and CosiMate tool suite [77] from Valiosys (previously Arexsys) are a commercialized version of the MUSIC design environment. While ArchiMate performs the subsystem and communication refinement, CosiMate provides the cosimulation facilities.

The result of the ArchiMate refinement is an architectural level representation of the system consisting of low level C code targeted to a specific processor for subsystems mapped to software and RTL VHDL for subsystems mapped to hardware. Based on this representation, the system can be synthesized using standard tools for software compilation and hardware synthesis.

Cosimulation at the system level is performed using CosiMate. However, CosiMate's performance at lower levels of abstraction is inadequate. Thus, ArchiMate provides a C++ model of the system based on a proprietary C++ HW library that allows a fast simulation of the generated system implementation at the architectural level. For clock-cycle accurate simulation, Arexsys provides an interface to the standard cosimulation environment Seamless CVE (see above).

**Cadence VCC** The Virtual Component Codesign (VCC) tool suite [11, 12] from Cadence provides a simulation-based design environment for heterogeneous hardware/software systems. Input to VCC is a functional system model including blocks imported from specification languages such as SDL, MATLAB, and SPW (DF process networks) as well as from implementation languages such as C, C++, and hardware description languages. Using cosimulation, this heterogeneous specification is functionally validated.

In contrast to the previous approaches, VCC clearly separates the functional specification from the target architecture of the system. Based on an extensive library support, the target architecture can be specified in terms of processors, buses, memories, and dedicated HW and SW blocks (e. g., RTOS). After defining a mapping of the functional blocks onto

the elements of the target architecture (including HW/SW partitioning), the performance of the system can be evaluated by simulation. In order to allow the rapid evaluation of possible system configurations (mappings), simulation accuracy is traded for simulation speed. Thus, the architectural elements are characterized at a high level of abstraction. Instead of relying on an HDL simulator for hardware blocks and an instruction set simulator for software blocks, VCC uses a performance modeling approach which allows the annotation of performance parameters to timing-free functional blocks. The performance parameters are estimated, e. g., for software blocks by performing profiling using a virtual instruction set characterized in terms of cycle delays for each instruction.

If a system configuration with an adequate performance level is found, the inter block communication is refined and corresponding interfaces are synthesized. Furthermore, a top-level netlist is generated in structural HDL based on the specified abstract target architecture model and C code targeted on the chosen RTOS/processor combination is generated for the software blocks. These implementation-level hardware and software descriptions can be used to generate a system prototype or implementation using existing synthesis tools. In addition, VCC generates scripts for a clock-cycle accurate cosimulation using tools such as Seamless CVE (see above). Mixed-level cosimulation where only certain parts of the system are simulated at the clock-cycle accurate level is supported as well.

## 3.3 Compositional Approaches

### 3.3.1 General Concept

While cosimulational approaches allow a systematic integration of the different subsystems of a heterogeneous specification, this integration via a communication protocol is fairly shallow [54]. Especially, it lacks sufficient information about the correlation of subsystems in order to allow a tool-based system-wide analysis and optimization.

*Compositional approaches* provide a deep subsystem integration by combining semantics of specification languages in a unified composition format. As shown in Figure 3.3, in a compositional approach the differently specified subsystems are translated into a composition format and merged to form a homogeneous representation of the complete system. Then, tool-supported system analysis and optimization is performed based on this homogeneous representation.

The key issue for a composition format is its representation efficiency with respect to analysis and optimization methods, i. e. while it is easily possible to provide a formalism to allow the representation of different models of computation in a single language, the composition of these models decides on the usability of the composition format. The composition is also used in order to classify the compositional approaches presented in the following into parallel and hierarchical composition.

### 3.3.2 Parallel Composition

Approaches based on parallel composition form the composition format by combining models of computation or elements of models of computation on a single hierarchical level.

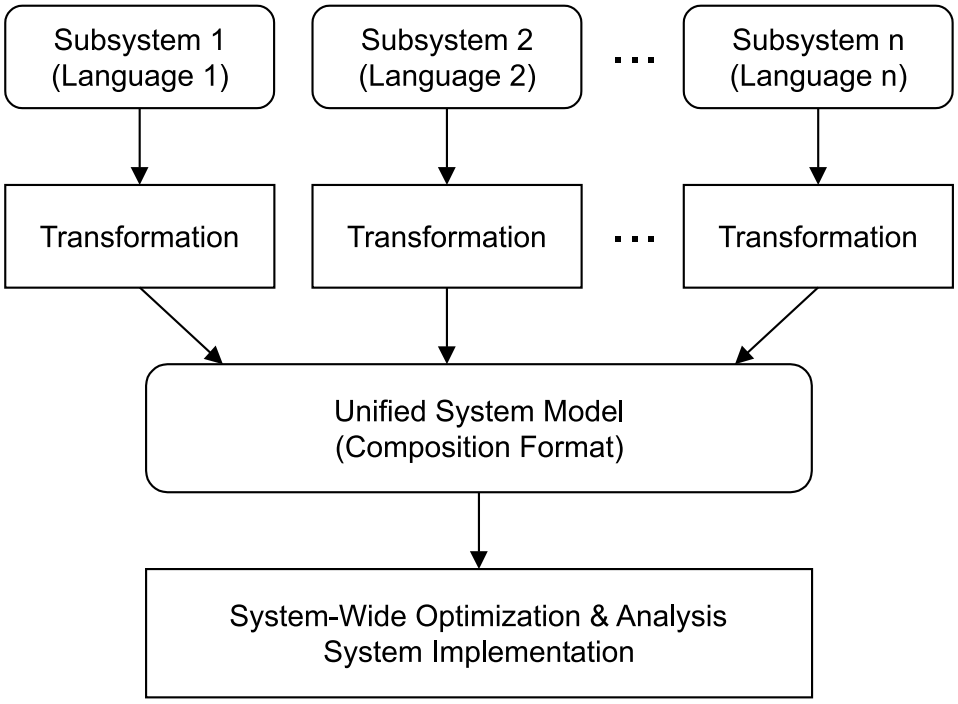



Figure 3.3: Generic design flow of compositional approaches

**Process Coordination Calculus PCC** An example for a parallel composition approach is the Process Coordination Calculus (PCC) [39]. PCC can be seen as a process network coordination language which allows arbitrary host languages in the range of the behavior specified by the coordination language.

In contrast to other process network coordination languages, PCC, however, defines two different process types: data-driven and event-driven processes. While a data-driven process *may* start if all of its input queues contain a sufficient amount of data (SDF semantics), an event-driven process has to start *immediately* if an event occurred at one of its input ports. Furthermore, three different unidirectional communication links are defined: streams, event queues, and registers. Streams are unbounded FIFO queues with destructive read and non-destructive write access and registers are one-place buffers with destructive write and non-destructive read access. Event queues have a more complex read and write access ensuring that each event is written and read exactly once. While event-driven processes operate on event queues and registers only, data-driven processes operate on streams, read from registers and write to event queues.

Although seemingly a new coordination language, PCC can be seen as a compositional approach of SDF process networks (subgraphs of data-driven processes) and synchronous/reactive process networks (subgraphs of event-driven processes). At the boundaries between event-driven and data-driven subgraphs, non-determinism in the sense of execution order dependent behavior arises. Examples are shown in Figure 3.4 where the execution order of data-driven processes $A$ and $B$ determines the sequence of the emitted events $\alpha$ and $\beta$ and thus the behavior of event-driven process $Z$. Similarly, the behavior of data-driven process $C$ depends on the register value set by process $Z$ and thus on whether $Z$ is executed before or after $C$. In order to enforce a deterministic network behavior,

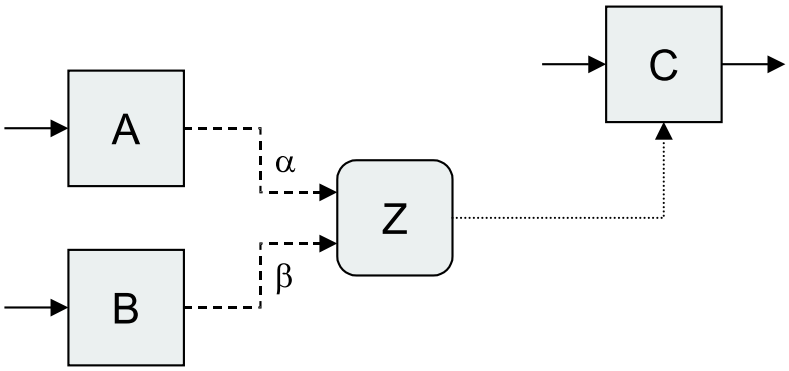



Figure 3.4: Part of an example PCC network

scheduling constraints (e. g., *execute A prior to B*) are specified determining a partial order of processes at 'domain' boundaries.

The complete elimination of non-determinism seems to be too restrictive as in some cases the non-determinism maybe desired or at least negligible. Furthermore, the scheduling constraints impose sequentialization constraints requiring synchronization that needs to be implemented in a run-time scheduler.

### 3.3.3 Hierarchical Composition

Hierarchical composition approaches are based on a composition format which combines different models of computation on several levels of hierarchy. They differ in the supported models but also in the way hierarchically higher models influence the behavior of lower level models.

**Ptolemy II** Ptolemy II [22] is an environment for heterogeneous concurrent modeling and design of embedded systems developed at the University of California at Berkeley. Ptolemy II allows the specification of functions in terms of actors and provides domains implementing the interactions of actors by enforcing the adherence to certain well-known models of computations (e. g., SDF, Kahn process networks, synchronous/reactive, FSM). Actors as well as domains are modeled in Java.

In terms of the established process network notation, domains provide the coordination language while actors are modeled in either Java (essentially the leaf host language) or are hierarchically refined by another process network with a possibly different domain. Ptolemy II allows the specification of arbitrarily deep and heterogeneous hierarchies.

Starting from such a hierarchical specification, Ptolemy II performs a hierarchical code generation, i. e. the code generation for a process network at one level of hierarchy provides the actor definition for the parent actor (at the higher hierarchical level). This can be seen as an upward propagation of model properties such as timing or communication. For example consider an actor $A$ refined by an FSM which has two states where each state is refined by a different SDF graph as shown in Figure 3.5. As the transitions of the FSM have no guarding conditions, the FSM performs a transition at each firing of $A$. Thus, actors $A_1$ and $A_2$, refining states $S_1$ and $S_2$, respectively, are alternately exe-

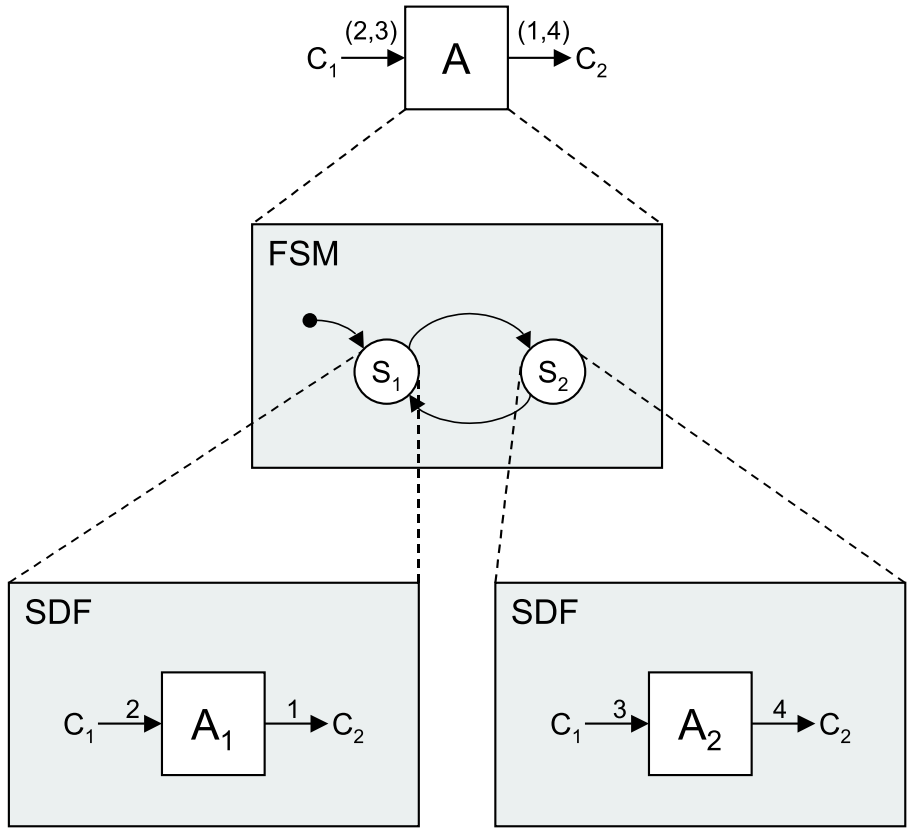



Figure 3.5: Example hierarchical process network modeled using Ptolemy II

cuted. Then, actor $A$ has two different communication behaviors depending on the state of its refining FSM essentially showing cyclo-static behavior [5]. Thus, as a result of the property propagation, the three level hierarchy can be replaced by a cyclo-static actor with two communication behaviors. This is an example for the optimizations performed by Ptolemy II during code generation. Targets for code generation are simulation acceleration (compiled simulation) as well as embedded software and behavioral hardware description.

An interesting subclass of Ptolemy II is the *charts model [34] combining finite state machines (FSM) and a variety of concurrency models (CM: dataflow, discrete-event, and synchronous/reactive) in an alternating hierarchy ($\ldots CM - FSM - CM - FSM - \ldots$). Thus, as already seen in the example, FSM states can be refined by concurrency models and concurrency model nodes (e. g., dataflow actors) can be refined by FSMs. In summary, *charts models concurrent finite state machines with a variety of concurrency semantics.

Restrictions of higher level domains impose constraints on the hierarchical refinements of actors or states in Ptolemy II. For example, actor $A$ in Figure 3.5 can not be embedded in an SDF domain as its communication behavior is not fixed. Similarly, the FSM domain requires transitions to have a finite reaction time. With states being refined by, e. g., cyclic SDF graphs which may never terminate, the finite reaction time has to be enforced by restricting concurrency models to perform a single *iteration* per activation. For SDF graphs, this iteration is defined by the minimum set of actor firings which return

the FIFO queues to the state at the beginning of the iteration. This set is also called a minimum cycle. In contrast to synchronous FSMs, the Ptolemy II FSMs thus have a transition timing which may even be state-dependent.

While being very efficient and elegant for systems specified within the environment, Ptolemy II is less suitable when it comes to incorporating system parts without systematic model of computation such as legacy code as these components do not provide rules for the upward propagation of properties. Furthermore, Ptolemy II requires precise and complete modeling which decreases its applicability at early design stages or at system integration.

**FunState** The FunState internal design representation [88] supports the explicit separation of data and control flow in subsystems called components. These components consist of a dataflow network and a finite state machine controlling the elements of the dataflow network. As the dataflow network may contain other components, FunState provides a hierarchical composition of FSMs and dataflow networks. The difference to *charts, however, is the fixed concurrency model (communication via FIFO queues and registers) and the fact that the FSM directly controls the actors of the dataflow network instead of the dataflow network as a whole. That means that an FSM transition can explicitly trigger the execution of a single dataflow actor instead of triggering the execution of a minimum cycle of the dataflow network. This fine control granularity makes FunState an excellent choice for the representation of scheduling mechanisms and the intuitive visualization of state-based process behavior.

FunState is intended as an internal design representation allowing to capture subsystems described with different models of computation. In [87], a timed version of FunState and symbolic methods for analysis as well as scheduling of FunState representations are presented.

**CoCentric System Studio** Originally called El Greco [9], CoCentric System Studio [89] is a system-level design environment based on a combination of control models (FSMs) and dataflow models (dataflow process networks). Besides allowing arbitrary hierarchical nesting of different models similar to Ptolemy II, it also features the combination of control and dataflow models at the same level of hierarchy. Here, composition types are AND (concurrent execution), OR (sequential execution), and GATED (mutual exclusive execution with state preservation).

Still, instead of the well-defined and restrictive integration semantics of *charts, CoCentric System Studio allows a variety of termination semantics of nested models. This multitude of possibilities diminishes the analysis capabilities of the model in comparison to *charts. However, the semantic variety can support the representation of target architecture influences in the system description. Thus, CoCentric System Studio provides rather an implementation-level language than an abstract specification language.

## 3.4 Summary and Conclusion

In this section, the strengths and weaknesses of cosimulational and compositional approaches to multi-language embedded system design are discussed and open problems

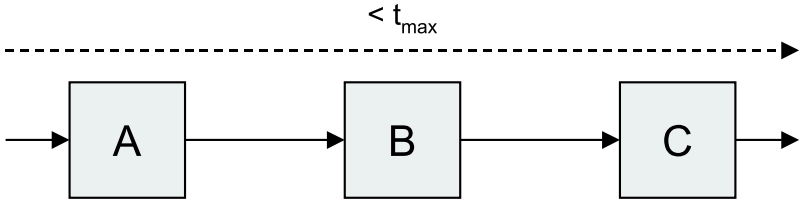



Figure 3.6: Example of a system-level timing constraint spanning three different subsystems

are identified.

Cosimulational approaches provide a flexible and systematic integration for heterogeneously specified systems. Essentially, the cosimulational integration can also be seen as a process network with the backplane protocol as coordination language and the different specification languages as host languages. This clear separation of subsystem coupling and subsystem specification leads to the two key strengths of the cosimulational approach:

- the reuse of the language-specific design environments and the resulting support of domain-specific optimization and analysis as well as
- the generality to include all kinds of subsystems as long as they can be executed or simulated.

This generality, howewer, comes at a price. The relatively shallow integration of the different subsystems does contain very little information of inter-subsystem correlations such as correspondences of subsystem states. This information is not needed for cosimulation or interface synthesis as the correlation is inherent in the communicated data. Yet, it significantly limits system-wide analysis and optimization. While manual analysis and optimization may be possible as system architects usually know about subsystem correlation, tool-based methods are inhibited by the lack of a coherent formalism to represent the correlation information at an abstract level.

Yet due to the system complexity, tool-based system-wide analysis and optimization is very important in order to allow a safe integration and a globally optimized implementation of heterogeneously specified embedded systems. An example for problems requiring system-wide analysis is an end-to-end timing constraint across more than one subsystem as shown in Figure 3.6. Although a bound on the constrained system response time can be obtained by adding the individual worst case response times of subsystems $A$, $B$, and $C$, this bound may be too conservative as, e. g., subsystems $A$ and $B$ may never both show their worst case response times at the same time. Too conservative bounds result in overimplementation i. e. in the provision of a too powerful and costly target architecture. A typical optimization strategy that is absolutely necessary to obtain an efficient implementation is resource sharing. In order to facilitate optimized resource sharing across language or subsystem boundaries, a system-wide analysis of the respective subsystem properties (e. g., process activation for sharing of processing elements) is required. Otherwise, a fall back is to resort to scheduling methods based on time slices (e. g., [62]) and non-overlapping memory allocations. In both cases, application-specific optimization potential is not considered.

(Co-)simulation can only reliably validate system properties in order to allow resource sharing or verify the satisfaction of constraints if being exhaustive. However, exhaustive simulation is generally infeasible (especially in the presence of uncertain environment timing) such that simulation yields an unknown coverage of corner cases resulting in worst cases for system properties such as, e. g., a maximum response time [96]. Thus, formal methods for the system-wide analysis and optimization are needed.

A requirement for formal analysis and optimization is the representation of subsystem correlation information at a higher level of abstraction. This is provided by the deep integration of different subsystems by compositional approaches which create a coherent formalism for the representation of the complete system. The efficiency of this composition format with respect to analysis and optimization methods decides on the usefulness of the approach.

The strength of this deep integration causes also the greatest weakness of most compositional approaches, the restriction to a limited subset of specification languages. Apart from FunState having been developed as an internal design representation, the presented approaches all facilitate direct specification in their composition format with limited facilities to consider or import system parts specified outside of the respective approach. Particularly problematic is the representation and integration of subsystems with only partially available information such as legacy code or an incomplete specification. Thus, these approaches rather promote a ”one-tool-chain-approach” instead of the flexible ”multi-tool-approach” facilitating the reuse of the existing domain-specific design environments as advocated by the cosimulational approaches.

Following from this discussion, an ideal multi-language design approach should integrate aspects from both cosimulational and compositional approaches. On the one hand, it should support the incorporation of subsystems with incomplete information as well as the reuse of design environments and libraries. On the other hand, it should enable the formal analysis and optimization of the system. In the following chapters, a compositional approach to embedded system design is presented which combines the flexibility of a truly multi-language specification with the analysis support of a unified internal design representation. This approach is the System Property Interval (SPI) workbench.

# Chapter 4

# The SPI Model

This chapter describes the SPI model, an internal system representation targeted to facilitating system-wide analysis and optimization. The SPI model has been developed in a cooperation of the Technical University of Braunschweig, Germany, the Swiss Federal Institute of Technology Zurich, Switzerland, and the University of Paderborn, Germany. After introducing and motivating its basic modeling concept in Section 4.1, the structure of the SPI model (Section 4.2) and the constructs defining the properties of the model elements (sections 4.3 to 4.5) are presented. Based on the presented model elements, the SPI execution model is detailed in Section 4.6. Before concluding the chapter with a summary, model extensions to represent function variants and the system environment including constraints are presented.

## 4.1 Modeling Concept

Following from the discussion of multi-language design approaches in Chapter 3, the overall motivation for the SPI model is to allow the reliable validation of non-functional system properties while being sufficiently flexible to allow the representation of various specification languages and system parts without systematic model of computation such as legacy code or partially specified components. This reflects the intended use of the SPI model as internal design representation for the SPI workbench relying on a true multi-language specification and cosimulation for functional validation.

Thus, SPI is no novel specification language. Rather, SPI captures the information relevant for system-wide analysis and optimization from a multi-language system specification while abstracting the exact functionality. This abstraction can be motivated by the fact that from a resource sharing perspective the properties of interest are not the exact function of a process but the time the process requires a certain resource for its computation, the time needed for communication, and the memory needed for storing communicated values.

The basic principle of the SPI model is to assume uncertainty of all properties such that the default behavior yields unbounded process execution times or data production and consumption. This uncertainty can be limited using the SPI model constructs. This principle allows the explicit specification of information even if only partial information is available.

The function abstraction as well as the uncertainty assumption make SPI a non-executable model. Rather, a SPI representation of a system bounds all possible behaviors of the respective system. An additional consequence from these concepts is that a check for the correctness i. e. consistency of a model may not always be possible. An example for this will be shown in Section 6.1.1. However, due to the intended use of the SPI model as an *internal* design representation generated by tools rather than manually, it seems suitable to assume correct representations.

From the intended use of the SPI model as internal design representation furthermore follows that, during the development of SPI, compactness in the sense of few model constructs has been stressed in comparison to specification comfort. However, in Section 4.4 an example is shown on how to define ”comfort” functions on top of basic model constructs.

Due to the requirement to represent complex embedded systems, the SPI model provides constructs to model the system coherently with its embedding environment including imposed constraints. In order to cope with the system complexity, the SPI model furthermore supports the modeling at various levels of detail. This concerns the granularity, i. e. the amount of computation performed by a process, as well as the modeling accuracy or abstraction, i. e. how accurately the behavior of a process is represented.

## 4.2 Basic Model

Like many of the models of computation presented in Chapter 2, the SPI model is based on the notion of process networks. SPI processes are functional in the sense of Kahn [58] meaning that the output sequences are a function of the input sequences. In other words, processes may have local data and therefore may have and manipulate a state.

Communication between processes is modeled by the exchange of data tokens via unidirectional channels. Implicit, global communication is not allowed, i. e. inter-process communication has to be explicitly modeled using channels. There are two types of channels:

- **Queues** are unbounded FIFO-ordered buffers. Read accesses are destructive (i. e. read tokens are removed from the queue) while write accesses are non-destructive (i. e. written tokens are appended to the queue). This ensures a causality and synchronization between sending and receiving processes in the sense that data can not be read before it has been written and before all data written earlier to the queue has been read. The destructive read semantics also exclude a test for input data without reading it i. e. removing it.

- **Registers** are single-place buffers. Read accesses are non-destructive (i. e. the same data token can be read more than once) while write accesses are destructive (i. e. a written token replaces the token previously stored in the buffer). Register accesses are unsynchronized in the sense that processes may read data from a register regardless of data written to the register and data may be overwritten before being read.

All channels have at most one writing process. Otherwise, processes writing to the same channel would have to be synchronized in order to enforce deterministic network behavior.

While registers may have more than one reading process (multi-reader registers), queues are constrained to have at most one reading process. This restriction prevents process activation conflicts due to the destructive read semantics and the data-driven activation principle described later in this section. Furthermore, channels do not have to have a reading or writing process. For example, channels without a writing process may contain initialization or customization data.

This basic model can be represented by a *model graph* in which processes and channels are denoted as nodes connected by edges. The model graph serves as a graphical representation of the SPI model and will be subsequently extended throughout this chapter by introducing model parameters and incorporating hierarchical refinement (cluster graph). SPIML, a textual representation of the SPI model based on XML, will be introduced in Appendix A.

**Definition 1** ***(Model Graph)***
*The model graph is a directed bipartite graph* $G = (P, C, E)$ *where*

- $P$ *denotes the set of process nodes,*
- $C = Q \cup R$ *denotes the set of channel nodes with* $Q$ *and* $R$ *being the sets of queues and registers, respectively,*
- $E \subseteq (P \times C) \cup (C \times P)$ *denotes the set of edges,*
- *for each* $q \in Q$, $indeg(q) = outdeg(q) \leq 1$, *and for each* $r \in R$, $indeg(r) \leq 1$ *with* $indeg$ *and* $outdeg$ *being functions which return the number of a node's incoming and outgoing edges, respectively.*

□

The constraints on the number of edges connected to a channel (item 4 of model graph definition) reflect the previously introduced restrictions on the number of reading and writing processes of a channel. In the graphical representation, queues are represented by circles whereas registers are represented as squares.

In the following paragraphs, a short overview of the SPI execution model is given in order to introduce the basic concepts needed for reasoning in the succeeding sections. The execution model is discussed in more detail in sections 4.5 and 4.6.

A process performs its computation sequentially and without functional interruptions, i. e. during execution a process will not wait, e. g., for input data to become available. Thus, process synchronization such as blocking communication has to be resolved by process activation. However, external interrupts of processes due to implementation decisions (e. g., by other processes having a higher priority) are possible but do not have to be modeled in the SPI model which is targeted at implementation-independent application specification.

Process activation is based on data availability, i. e. a process is activated if its required input data is present and *may* execute if it is activated. This activation principle is similar to dataflow process networks [69] and does not overconstrain the design space by enforcing a fixed timing throughout the network.

Both computation and communication may consume time. However, the amount of time has to be finite, i. e. process execution has to terminate in a bounded time. Otherwise, reliable and fair resource sharing or statements concerning system timing (e. g., the satisfaction of timing constraints) would be impossible.

Processes may communicate data tokens throughout their execution during so called *communication regions*. A communication region denotes for a connected channel the time interval during which the process may send data on or receive data from this channel. The concept of communication regions enables a flexible adaption of the process communication behavior in SPI, necessary to model several input languages and models of computation. A fixed communication scheme (e. g., read at start, write at end) is not flexible enough, e. g., to capture arbitrary software processes which can communicate at any time during their execution.

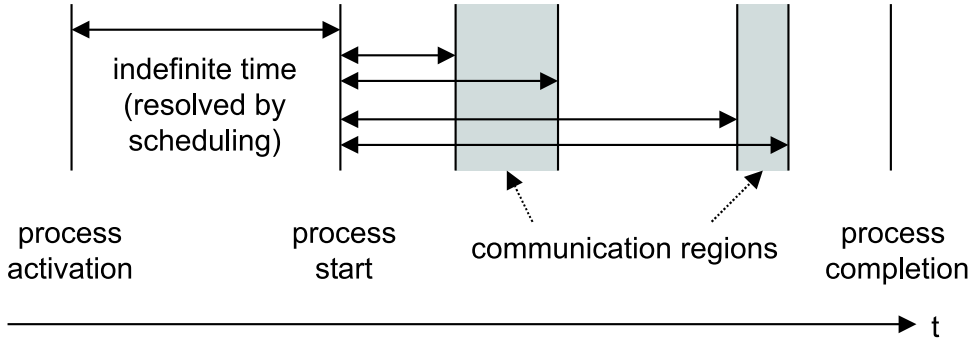



Figure 4.1: Qualitative timing of SPI process execution assuming non-preemptive implementation

The qualitative timing of a SPI process execution is shown in Figure 4.1. The time between process activation and start of process execution is indefinite (due to the *"may start"*-semantics) and is resolved during scheduling. The figure shows how communication regions are specified relative to the start time. The depicted timing diagram assumes non-preemptive implementation i. e. a process once started is never interrupted, e. g., by a higher priority process. However, this assumption is only made to clearly show the relation between process start, completion and communication regions and is not a restriction of the SPI model. In case of an interrupt, the execution time already passed until the interrupt simply has to be traced for analysis.

## 4.3 Parameters

After having defined the structure of the SPI model in the previous section, this section adds meaning to the SPI model by providing constructs to further describe the model elements. The main objective of the SPI model is to capture the information relevant for the validation of non-functional system properties. In order to determine system properties, the contribution of model elements to these properties has to be known. Typical properties of interest are related to resource requirements of an element and its interaction with its

environment (connected elements). In this context, the term *process behavior* includes the function or functionality of a process as well as its non-functional properties.

These properties of processes and channels are defined by parameters that are annotated to the corresponding model elements. This allows for an easy extensibility and adaptation of the SPI model to include all required information for a certain optimization goal or task in the design flow.

The parameters do not need to be set to a single value but can be specified using *behavioral intervals* denoting uncertainty of the exact parameter value. Using a behavioral interval, this parameter and thus the uncertainty of its value is constrained by an upper and lower bound.

Several sources of this uncertainty can be identified by considering the influences on the properties described by these parameters. They can be classified according to the following types:

1. **Abstraction of data-dependent functionality**
   The functionality of a process does often depend on its (input) data. Typical examples are `if-then-else` structures that choose one of two possible process execution paths depending on a condition based on certain data. Each process execution path may lead to a different execution behavior and thus to different values for the describing parameters. These different values can be merged to behavioral intervals yielding an abstraction of the overall process execution behavior.

2. **Limited analyzability or non-determinism of input language**
   Some input descriptions feature non-determinism e. g. to represent freedom in the specification that may be used for design space exploration. An example for such a non-determinism is a state with several concurrently enabled exit transitions as featured e. g. by the FunState model of computation [88]. An example for a limitation of analysis capabilities is the complexity of the determination of data dependencies and control flow in a C program heavily using C pointer arithmetic. In both cases exact values for parameters may not be determined resulting in behavioral intervals.

3. **Limited analyzability of target architecture**
   For parameters not only depending on the application but also on the target architecture, modern features of computer architecture such as caches, out-of-order execution, or data-dependent instruction costs inhibit the specification of exact values for parameters. However, these influences can be analyzed and bounded resulting in a behavioral interval.

4. **Incomplete specification**
   In order to obtain an idea of (sub-)system properties in an early design stage, a first design space exploration may be performed even before the complete (sub-)system functionality is fixed. Thus, parameters have to be estimated resulting in a range of possible values for a certain parameter that can be denoted as a behavioral interval.

The resulting behavioral intervals lead to a non-deterministic process behavior in the SPI model. However, the types of non-determinism differs. While for the uncertainty sources one to three the parameter may switch between all possible values of the interval at run-time, the non-determinism of the incomplete specification (source four) will be

eliminated during design time such that it can be assumed that the parameter will take just one of the possible values of the uncertainty interval at run-time. In the SPI model, however, the different types of non-determinism are not distinguished since the advantages for analysis and optimization methods possibly resulting from a utilization of this difference are minimal at most. The dominant reason for this is that source four is typically superposed by sources one to three.

Another possibility of non-determinism already captured in the specification is a choice of different functional alternatives. All of several alternatives yield a correct system behavior, and it is left to the optimization step to choose and implement one of the functional alternatives. This is fundamentally different from the concept of behavioral intervals where optimization and analysis has to assume the occurrence of all values in the behavioral intervals and may not choose just a single one. Thus in the SPI model, functional alternatives may be represented using the concept of function variants (see Section 4.7).

In the following, we distinguish parameters based on their dependencies on application and target architecture. Functional parameters are solely dependent on the application while implementation-dependent parameters also depend on the target architecture.

### 4.3.1 Functional Parameters

The value of a functional parameter depends only on the functionality of the corresponding model element and is independent of its implementation. Thus, it can be determined by analysis of the corresponding input language element only and does not change due to implementation decisions.

#### 4.3.1.1 Communication

The behavior of a process network is defined by the process behaviors and the communication between processes. It has been motivated above that not the detailed functionality but rather the non-functional properties of processes need to be known for the purpose of the SPI model. From neglecting functionality follows that the content of the communicated data may be neglected as well, since one evidently doesn't need to know the operand of a function if the function is unknown. Thus in the SPI model, communicated data is represented by its amount only and can be specified in terms of *data tokens* which are the atomic data unit in the SPI model.

The amount of data communicated by a process at each execution is of central importance for e. g. the dimensioning of communication resources and the determination of activation rates of subsequent processes. Therefore, a *data rate* parameter denoting the number of data tokens communicated on a certain channel at each execution is annotated to processes.

**Definition 2** ***(Data Rates)***
*Let* $Inputs_p = \{c \in C \mid e = (c, p) \in E\}$ *denote the set of input channels of process* $p \in P$ *and* $Outputs_p = \{c \in C \mid e = (p, c) \in E\}$ *denote the set of output channels of* $p$*.*

*For each input channel* $in \in Inputs_p$*, a process* $p$ *has an input data rate interval* $r_{in} = [r_{in,min}, r_{in,max}]$*, which constrains the number of data tokens read from channel* $in$ *per process execution, where* $r_{in,min} \leq r_{in,max}$ *and* $r_{in,min}, r_{in,max} \in \mathbb{N}_0^+$*. Analogously,*

*for each output channel* $out \in Outputs_p$*, there is an output data rate interval* $s_{out} = [s_{out,min}, s_{out,max}]$. □

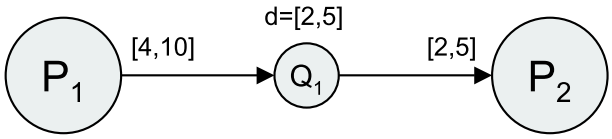



Figure 4.2: Two SPI processes communicating over a SPI queue with preassigned tokens

Figure 4.2 shows two communicating SPI processes. Process $P_1$ has an output data rate of $[4, 10]$ for channel $Q_1$, i. e. at each execution, $P_1$ will write at least $4$ and at most $10$ tokens to $Q_1$. Similarly, process $P_2$ reads between $2$ and $5$ tokens from queue $Q_1$ at each process execution. Note that in the graphical representation data rate intervals often appear without the corresponding parameter names for reasons of brevity. Similarly the indices are often omitted if the association of the parameter is clearly represented by its placement.

While data rate intervals are generally independent of the type of the corresponding channel, there is the following additional constraint on the maximum value of data rate intervals for registers $r \in R$.

$$r_{r,max}, s_{r,max} \leq 1$$

This restriction follows from the fact that registers are single-place buffers that contain exactly one token. This is no degradation of modeling capabilities since a shared memory communication between two processes that involves several variables which are written and read separately can be modeled using a separate register for each variable.

At system start-up, buffers are often initialized with predefined values. Typical examples are delays in synchronous dataflow [68] or the initial state in state-based models. In SPI, this translates to assigning a number of tokens to channels.

**Definition 3** ***(Channel Initialization)***
*Associated with each channel* $c \in C$*, there is the parameter* $d_c = [d_{c,min}, d_{c,max}]$ *denoting the initial number of data tokens on channel* $c$. □

An example for a channel initialization is shown in Figure 4.2. There, the channel initialization interval of $[2, 5]$ denotes that at least $2$ and at most $5$ data tokens are preassigned to queue $Q_1$.

For registers $r \in R$, $d_r$ is always equal to $1$. Again, this is due to the fact that a register is a single-place buffer that is accessed by non-destructive read. Thus, a register contains always exactly one token and is never empty.

So far, communicated and initial data have been modeled in terms of data tokens. In order to be able to determine absolute values for the amount of data as needed e. g. for the dimensioning of buffers, the size of data tokens has to be specified. Since data tokens are defined to be the atomic unit of data in the SPI model, they are indivisible and cannot be split in several smaller tokens or merged to a single larger token. This means that the

sending and the receiving process have to be compatible in terms of the size of their communicated tokens. Thus, the *token size* parameter is annotated to channels as it describes a property of the communication between two processes represented by a channel.

**Definition 4** ***(Token Size)***
*Associated with each channel $c \in C$, there is a token size $b_c = [b_{c,min}, b_{c,max}]$ denoting the size of each token communicated on channel $c$.* □

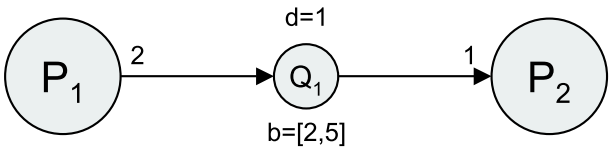



Figure 4.3: Example modeling the same system as depicted in Figure 4.2 but with an uncertain token size instead of uncertain data rates

An interval for the token size parameter can have different meanings. On the one hand, it may represent uncertainty of the size of communicated data due to an incomplete specification. On the other hand, it may also be used to model a communication of variable length packets. In this case, it is more favorable to model the variable amount of communicated data using uncertainty for the token size parameter than for the data rate parameters.

Consider the example of Figure 4.2 (uncertain data rates) and compare it to the modeling using an uncertain token size as depicted in Figure 4.3. Assuming a token size interval of $b_{Q_1} = [1, 1]$ for the modeling in Figure 4.2, the communicated amount of data in both representations is equivalent. However using the uncertain token size, it can be seen that for each execution of process $P_1$ process $P_2$ has to be executed twice in order to prevent a buffer overflow for $Q_1$. This information about the relative execution rate is lost for the representation using uncertainty of data rates since, e. g., it also includes the possibility that $P_1$ writes and $P_2$ reads exactly $4$ tokens at each of their executions resulting in $P_1$ and $P_2$ having the same number of executions. Note that, for brevity, behavioral intervals denoting a single value (lower and upper bound being equivalent) are typically represented by this single value only.

Due to registers being single-place buffers, the token size $t_r$ of a register $r$ also defines the size of the register. The queues are defined to be unbounded which is not implementable due to the possibly infinite amount of required memory. However, the maximum number of data tokens in a queue during run-time can be analyzed a priori for a given implementation using methods such as [36]. Multiplied with the token size, this yields the amount of memory required for an implementation of the queue that prevents a buffer overflow.

In this context, it has to be stressed that the token size parameter only captures the functional data, i. e. it does not consider possible encoding or protocol overhead resulting from a certain implementation of the channel. These influences are solely dependent on the target architecture and thus are captured by an architecture model instead of the SPI model. Thus, the token size parameter is classified as a functional parameter. Still, there is

a possible implementation dependency of the token size parameter. This is due to the use of data types that do not explicitly define the number of bits used for their implementation e. g. `integer` in ANSI-C. In this case, the number of bits used for their implementation and thus the size of a token containing one or more integer variables depends on the word length of the used processor. However, many languages for embedded system design (e. g. Simulink, SystemC) have fixed-length data types (e. g. `int32`) and also C code for embedded systems usually features `#define` statements to fix the number of bits for data types. Otherwise, the range of possible values for the toke size parameter has to be conservatively estimated.

#### 4.3.1.2 Virtuality

For modeling purposes, the concept of *virtuality* is introduced for processes and channels.

**Definition 5** ***(Virtuality)***
*Associated with each process and each channel* $n \in (P \cup C)$*, there is a virtuality flag* $v_n \in \{'true', 'false'\}$ *which denotes the fact whether the process or channel is part of the system to be implemented (*$v_n := 'false'$*) or has been introduced for modeling purposes only (*$v_n := 'true'$*).* □

Virtual model elements have the same semantics as non-virtual ones. However in contrast to non-virtual model elements, there is usually no direct implementation equivalent to a virtual model element. Rather do they visualize information that has to be regarded during implementation.

In the graphical representation, virtual model elements are denoted by dashed lines. Edges may also appear dashed although they do not have a virtuality flag. The reason for this is that edges are seen as part of a channel formed by a channel node and its two connected edges and channels are always depicted alike. Thus, if an edge is connected to a virtual channel the edge appears dashed.

In this section, the different possibilities to use virtual model elements are only briefly introduced together with a reference to the section that contains more detail. Virtual model elements are used to:

- model the system environment coherently with the system (see Section 4.8)
- represent other activation principles (e. g. periodic time-driven activation, see Section 6.1.2) based on SPI's data-driven activation
- represent constraints on design space (e. g. relative execution rates, see Section 6.1.1)

### 4.3.2 Implementation-Dependent Parameters

System properties like timing, memory requirements, or power consumption depend not only on the application but also on the target architecture the application is mapped on. There are several possibilities how to represent this additional dependency.

1. **Set of behavioral intervals**
   For an implementation-dependent parameter, a set of behavioral intervals is annotated to a model element. Then, each behavioral interval represents the possible

values for the parameter if the model element is mapped to *a certain* possible resource. With any taken design decision (e. g. allocation), a subset of these intervals is selected.

2. **Single behavioral interval**
   The behavioral interval for an implementation-dependent parameter is the superset of the intervals as described in the previous item. Then, the behavioral interval represents the possible values for the parameter with respect to *all* possible resources. With any taken design decision, then the interval gets narrower.

3. **Extended graph structure**
   The SPI model is kept implementation-independent and is augmented with a model of the HW/SW target architecture and edges between elements of both models representing possible mappings of SPI model elements to architectural resources. Then, the behavioral interval for an implementation-dependent parameter with respect to a certain resource is annotated to the mapping edge between the resource and the corresponding SPI element. With any taken design decision, then mapping edges are selected.

The third possibility is clearly the most elegant of the three. Its key advantage is that the placement of the behavioral intervals naturally represents both dependencies on application and implementation. This is regarded in the current implementation of the SPI workbench where implementation-dependent parameters are placed on mapping edges between SPI model elements and elements of an architectural model [64]. However, the relation between target architecture components and SPI elements are far from trivial in the general case including e. g. preemptive scheduling. This is ongoing work. Thus in the context of this work, implementation-dependent parameters are annotated directly to SPI model elements in the meaning of possibility two.

Implementation-dependent parameters of SPI model elements can be determined by analyzing or estimating the parts of the implementation that correspond to the model element with respect to the described property. A method to obtain behavioral intervals for software processes is described in Section 6.2.1. Note that implementation-dependent parameters of virtual model elements are not defined since these elements do not have a direct implementation equivalent. However in the following sections, influences of the virtuality of model elements on implementation-dependent parameters will be shown.

#### 4.3.2.1 Timing

The correctness of embedded systems often does not only depend on the result of the computation but also on the time the result is made available. These systems are referred to as real-time embedded systems. A simple example is an automotive airbag control system that in case of a detected impact has to not just release the airbag but release the airbag within a certain specified time.

In order to validate the timely release of the airbag the execution time of the control system has to be known. Evidently, the time needed for the execution of a system does not only depend on the function to be performed but also on the implementation of this function, i. e. the resource(s) the function is performed on. For example, the time to

perform a Fast-Fourier-Transformation on a digital signal processor or dedicated hardware is a lot shorter than on an 8-bit-microcontroller.

The overall execution time of the system depends on the execution times of its components. In the context of the SPI model these components are processes and channels. Due to limited analyzability of the target architecture, the execution time of processes is rarely a constant value but rather can be specified by an behavioral interval. System-level timing analysis methods (for a comprehensive overview see [29]) often use only the upper bound of this interval, the so-called *worst-case execution time (WCET)*, since they typically only consider the validation of deadlines or maximum timing constraints. However, a general representation as the SPI model needs to consider not only the upper but also the lower bound of the execution time interval for two reasons. Firstly, also minimum timing constraints (e. g. system may not answer before bus idle time is over as defined by protocol) have to be accounted for and, secondly, for distributed systems the lower bound on the execution time has to be considered even for maximum timing constraints due to the so-called scheduling anomalies (e. g. [32]). Thus, a *latency* parameter denoting the execution time is annotated to SPI processes.

**Definition 6** ***(Process Latency)***
*Each process $p \in P$ has a latency time interval $lat_p = [lat_{p,min}, lat_{p,max}]$ where $lat_{p,min}$ [$lat_{p,max}$] denotes the lower [upper] bound on the execution time of process $p$. The execution time is defined as the time between process start and completion assuming the process has exclusive resource access.* □

A complete timing description of a process execution does not only have to specify how long the process executes but also when during its execution the process communicates with its environment. This is important e. g. for buffer dimensioning where the earliest possible times a process writes to a buffer and the latest possible times a process reads from the buffer need to be known. Thus, *communication regions* are defined denoting the time intervals during which the process accesses its connected channels.

**Definition 7** ***(Communication Region)***
*For each connected channel $c \in (Inputs_p \cup Outputs_p)$, a process $p \in P$ has a communication region $cr_c = [cr_{c,min}, cr_{c,max}]$ where $cr_{c,min}$ [$cr_{c,max}$] denotes the earliest [latest] time process $p$ accesses channel $c$. The values of the bounds for a communication region are specified with respect to process start and assuming non-preempted process execution.* □

Similarly to the latency of a process, its communication regions are implementation-dependent. Intuitively, as the latency of the whole process varies for different resources, also the latencies of process parts separating different channel accesses and thus the time intervals, during which these channel accesses may occur, vary.

Again, although both latency and communication region parameters are defined for non-preempted process execution this does not exclude preemptive implementation of SPI processes. Rather, this parameter definition reflects a separation of concerns between timing analysis of a single process and timing analysis of a system (process network). In this context, process timing analysis assumes exclusive resource access and yields a core execution time interval (the SPI process latency) which is independent from the other processes of the system. Based on the computed core execution times, system timing analysis

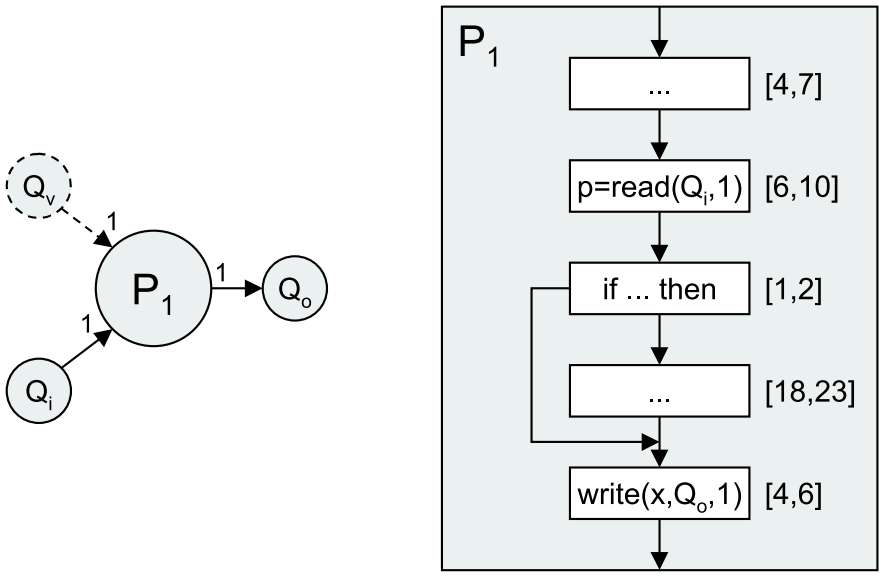



Figure 4.4: SPI process with connected channels and internal process control flow graph with annotated latency intervals

then considers the interaction and resource sharing influences (e. g. preemption delay) and computes a process response time interval based on the computed core execution times. A more detailed description of this two-level analysis approach can be found in [84].

The following example shows the use of the parameters concerning process timing. Figure 4.4 shows a simple SPI process ($P_1$) and its internal control flow graph. Annotated to each node of the control flow graph are the corresponding latency time intervals, i. e. the minimum and maximum time needed for the execution of this node. Now the latency interval of process $P_1$ can be computed by adding up the lower node latency bounds along the shortest path and the upper bounds along the longest path. This leads to a latency interval

$$lat_{P_1} = [(4 + 6 + 1 + 4), (7 + 10 + 2 + 23 + 6)] = [15, 48]$$

specifying that the non-preempted execution of process $P_1$ takes between $15$ and $48$ time units.

Similarly, the communication regions $cr_{Q_i}$ and $cr_{Q_o}$ can be determined by finding the shortest time between process start and start of the communication node resulting in the lower bound and the longest time between process start and completion of the communication node resulting in the upper bound. This evaluates to

$$cr_{Q_i} = [4, (7 + 10)] = [4, 17]$$

$$cr_{Q_o} = [(4 + 6 + 1), (7 + 10 + 2 + 23 + 6)] = [11, 48]$$

such that e. g. $cr_{Q_i}$ denotes that the reading access of process $P_1$ from channel $Q_i$ is performed between $4$ and $17$ time units after the process execution starts. It should be noted that $P_1$ is just used to illustrate the concept of communication regions. The general problem of process timing analysis requires more sophisticated approaches to cover loops and dependent branches. An example is given in Section 6.2.1.

In the description of the example so far, the virtual channel $Q_v$ has not been considered as it has to be treated differently than the non-virtual channels. The communication regions for those channels can be calculated by analyzing the implemented host code of the process (as seen in the example). However, virtual channels have no direct implementation equivalents (no access to $Q_v$ appears in the control flow graph) and thus their communication regions can not be analyzed. But since virtual channels are used to model specification information (e. g. timing constraints), it is important to be able to define the point in time of the process execution to which a constraint modeled by a virtual channel refers. Thus, for virtual channels the communication region parameter denotes a reference that can be used to determine the corresponding communication regions rather than a value interval.

**Definition 8** ***(Communication Region for Virtual Channels)***
*For each connected virtual channel* $c \in \{c \in (Inputs_p \cup Outputs_p) \mid v_c = 'true'\}$*, the communication region of a process* $p \in P$ *denotes a reference*

$$cr_c \in (\{'START', 'COMPLETION'\} \cup \{cr_c | c \in (Inputs_p \cup Outputs_p) \mid v_c = 'false'\})$$

*which defines the time interval at which process* $p$ *accesses channel* $c$ *during its execution. Meaningful references are all points in time or time intervals that are visible to the process environment, i. e. the start of process execution (*$'START'$*), the completion of process execution (*$'COMPLETION'$*) and the communication regions of all connected non-virtual channels.* □

For virtual channels, the communication region parameter is actually implementation-independent and thus belongs to the group of functional parameters. It is introduced in this section since it does not make sense without implementation-dependent parameters. In the example of Figure 4.4, the virtual channel $Q_v$ is introduced to constrain the start of process $P_1$. Thus, the communication region of $P_1$ for channel $Q_v$ is denoted by $cr_{Q_v} = 'START'$.

The SPI model captures the application influences on non-functional system properties. With the latency and communication region parameters, the influences of the process functionality on process execution timing can be specified. Although these parameters are implementation-dependent, they nevertheless abstract the process functionality in terms of its non-functional properties, here timing. For system-level timing analysis, these parameters are combined with purely architectural parameters such as task switching times to allow a complete description of the execution timing.

Analogous to computation, communication in the SPI model takes time as well. Yet, no additional parameter has to be defined in order to capture the application influences on communication timing. This is due to the fact that the functionality of communication is implicitly defined to be the identity operation. Thus, the equivalent of the latency or core execution time of a process is a purely architectural parameter and independent of the application. This is the transmission delay which defines the time needed to transmit a certain amount of data over a communication resource. Together with the token size parameter of a SPI channel and other communication resource specific parameters (e. g., packet size or initiation delay), the latency of a communication of a certain number of data tokens over this channel can be calculated using methods as described in [61].

#### 4.3.2.2 Power Consumption

Power consumption is a critical property for many embedded systems. This is certainly the case for mobile systems (e. g. cellular phones) where power consumption is directly related to the operating time due to a limited amount of battery capacity. But also for wired systems, power consumption can be critical in terms of practical limitations due to the dissipated heat or in terms of energy costs and reliability.

Clearly, power consumption depends on the target architecture as it is strongly influenced by architectural parameters such as supply voltage or clock frequency. However, power consumption does also depend on *which* functional units of the target architecture have to be used *for how long* to perform an application. Thus, power consumption is an implementation-dependent parameter.

The power consumption of a system varies widely with the performed tasks, as shown e. g. by the fact that the standby time of a cellular phone is typically up to fifty times higher than the talk time. Thus, for a meaningful analysis of system properties related to power consumption it is important to know the power consumed by the execution of different functions of an application. In the SPI model, these application functions are represented by processes and channels. Thus, a parameter capturing their power consumption is annotated to SPI processes.

**Definition 9** ***(Power Consumption)***
*Each process $p \in P$ has a power consumption interval $pow_p = [pow_{p,min}, pow_{p,max}]$ where $pow_{p,min}$ [$pow_{p,max}$] denotes the lower [upper] bound on the power consumed during one execution of process $p$.* □

Analogously to the latency parameter, the parameters capturing the power consumption of communication are purely architectural. Again, the only application-dependent parameter needed is the token size of a channel.

#### 4.3.2.3 Memory

When selecting a suitable target architecture for the implementation of a system, not only the performance level but also the provided memory capacity of an architecture is an important criterion. Thus, the memory properties of an application need to be captured just as its performance and power properties are by the parameters introduced in the previous sections. The memory properties of an application have in turn an impact on the timing properties by determining whether the instruction code of a process fits in the cache and on the power properties as memories also consume power.

In contrast to the memory properties of communicated data where a possible implementation dependency due to data types with a variable bit-width is usually eliminated, the memory properties of processes are inherently implementation-dependent. An important reason for this is the varying code density of microprocessors, i. e. depending on the instruction set of a processor the size of the code implementing the same application varies considerably. As usually a compiler is used for the creation of the instruction code from a higher-level description of the process, the code size and thus the memory properties of a process do not only depend on the target architecture but also on the design process as the quality and optimization strategy of the used compiler also effects the code size.

While the memory properties of the communication aspects in SPI have already been defined by the functional parameters concerning the number and size of communicated data tokens (see Section 4.3.1.1), the memory properties of processes are captured in the SPI model by a static and a dynamic memory size interval.

**Definition 10** ***(Memory Size)***
*Each process $p \in P$ has a static memory size interval $mem_{p,stat} = [mem_{p,stat,min}, mem_{p,stat,max}]$ where $mem_{p,stat,min}$ $[mem_{p,stat,max}]$ denotes the lower [upper] bound on the memory size needed for the allocation of process $p$ if it does not execute.*

*Furthermore, each process $p \in P$ has a dynamic memory size interval $mem_{p,dyn} = [mem_{p,dyn,min}, mem_{p,dyn,max}]$ where $mem_{p,dyn,min}$ $[mem_{p,dyn,max}]$ denotes the lower [upper] bound on the additional memory size needed for the execution of the process.* □

The memory model assumed by this parameter definition is that the memory size needed by a process consists of a time-invariant component ($mem_{p,stat}$) capturing instruction code and statically allocated data like constants or state variables and a dynamic component ($mem_{p,dyn}$) capturing the additional memory demand during execution including temporal variables and loop counters. Thus during execution, the memory size of process $p$ is the sum of both components ($mem_p = mem_{p,stat} + mem_{p,dyn}$).

This memory model consisting of only two parameters may be too coarse-grain for some analysis tasks. For example, the influences of instruction and data memory can be separated or the resolution of the development of the memory size over time can be increased e. g. by considering not only process start and completion but also communication regions. However, the easy extensibility of the SPI model allows the introduction of a more sophisticated memory model if needed.

#### 4.3.2.4 Other Properties

The previously described non-functional system properties timing, memory, and power consumption all have a direct application dependency, i. e. the execution of a SPI process takes a certain time, needs a certain amount of reserved memory, and consumes a certain amount of power.

Other non-functional system properties like size, weight, and cost depend primarily on the target architecture. There is only an indirect dependency on the application based on the fact that the application demands a certain performance level that has to be satisfied by the selection of a suitable target architecture. The elements of the target architecture including operating system or intellectual property components then determine the size, weight, and cost of the system. Due to the missing direct dependency on the application, these properties are not captured by any parameters of the SPI model. Rather they will be part of the architecture model augmenting the SPI model.

Currently, properties related to a hardware implementation of SPI elements like chip area are not captured since the focus in the area of analysis and optimization has been set on embedded software. However, the SPI model is extensible and can be easily adapted to capture additional properties if required by new implementation and analysis methods or optimization goals.

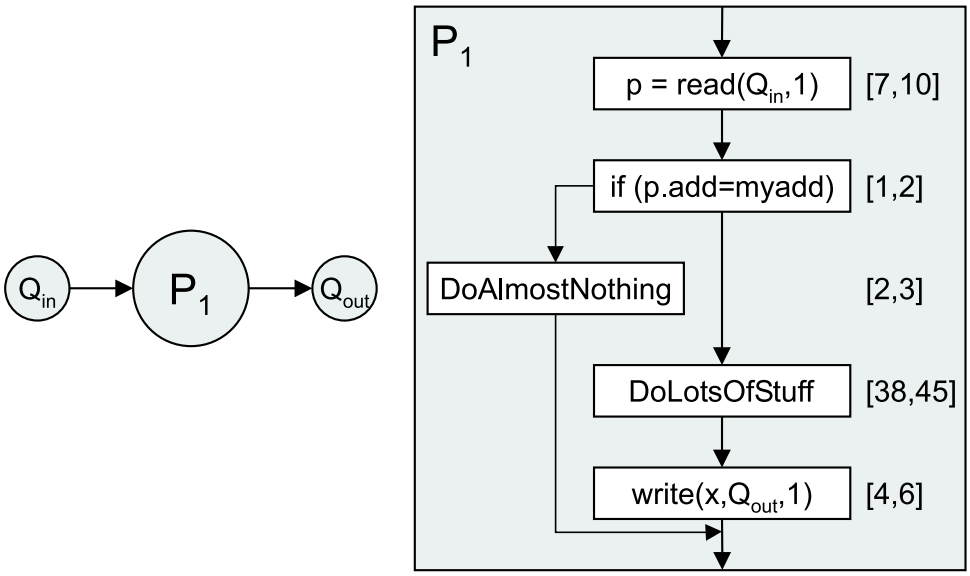



Figure 4.5: Packet receiver process $P_{rc}$ and corresponding control flow graph with annotated latency intervals

## 4.4 Process Modes

With the constructs presented so far, a SPI process has a single behavior described using uncertainty intervals. One of the origins of these behavioral intervals is the abstraction of data-dependent behavior, i. e. the unified representation of different process execution behaviors by merging their respective parameter values to a common interval. The following examples will show that there are situations where it is favorable not to merge but rather distinguish the different execution behaviors of a process.

Figure 4.5 shows the simplified process control flow graph of a packet receiver. Each node in the graph has annotations regarding its influence on the process latency time intervals (concerning a certain implementation), i. e. it takes between $38$ and $45$ time units to execute the node called *DoLotsOfStuff*. The communicated tokens are denoted by the second parameter in the `read()` and `write()` functions. Clearly, the process has two different execution paths that are chosen depending on the address of the received packet. While the first path featuring the *DoLotsOfStuff* node has a latency interval of $[50, 63]$, the second path including the *DoAlmostNothing* node has a comparatively small latency interval of $[14, 21]$. When both paths are merged to a single behavior, this behavior is characterized by the wide interval of $[14, 63]$ leading to the worst-case assumption that each process execution may take up to $63$ time units.

When having additional information denoting that packets arrive every $50$ time units with a maximum of $3$ out of $10$ consecutive packets being addressed to this receiver, the maximum utilization of the receiver can be calculated. Assuming packet buffering and a single non-pipelined processing element, this yields for the single interval representation

$$\frac{63 \times 10}{50 \times 10} = 1.26$$

suggesting that the chosen implementation is not capable of processing this packet stream.

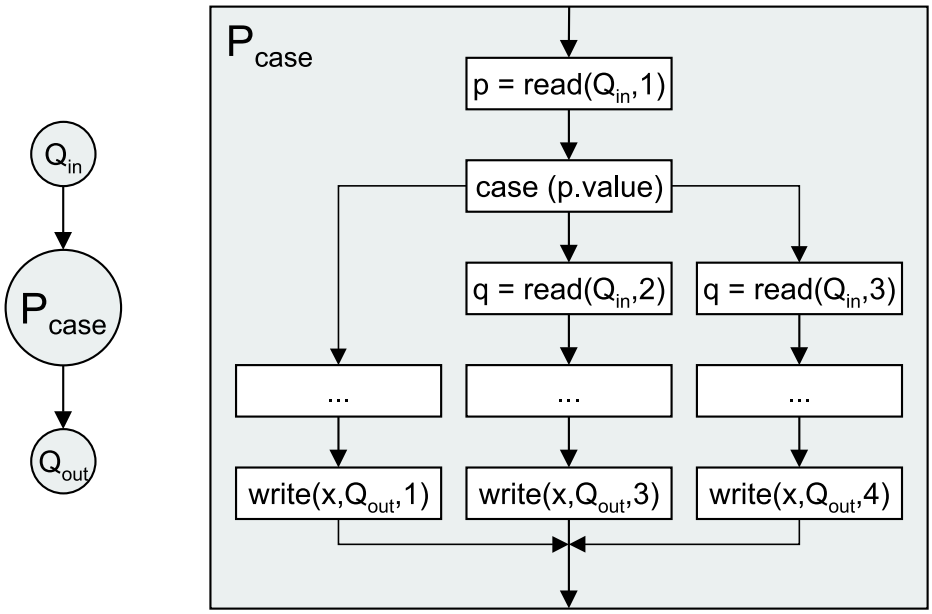



Figure 4.6: Example process with corresponding control flow graph

However, when both execution paths are distinguished the utilization evaluates to

$$\frac{(63 \times 3) + (21 \times 7)}{50 \times 10} = 0.672$$

showing that the implementation provides sufficient performance to process the packet stream correctly. Thus, for this example the single interval representation is too imprecise and leads to false rejection of a valid implementation.

The following example shows how correlations between values of different process parameters can be lost due to the use of behavioral intervals which may influence the quality of analysis results as well. Figure 4.6 shows the control flow graph of a process having three different execution paths. Depending on the execution path, the process consumes, processes, and produces $1$, $3$, or $4$ data tokens respectively. A representation of this behavior using single intervals leads to input and output data rate intervals of $[1, 4]$. Then however, the correlation between input and output data rate values is lost such that an analysis has to consider e. g. a process behavior of consuming $1$ and producing $4$ tokens although this will never occur. This is problematic since, due to the data-driven activation principle of SPI, produced tokens lead to activations of the process reading these tokens. Thus, the analytical possibility of a consume-1-produce-4 behavior leads to a maximum execution rate of the reading process (and a maximum inferred computational load) that is four times higher than the correct value. Thus in this case, the single interval representation may be too imprecise for an efficient analysis since it does not capture the correlation between different process parameters.

A consequence from these examples is to provide a means to distinguish and explicitly represent different execution behaviors of a process. This means is the concept of *process modes*. As an execution behavior is described using process parameters, a process mode is defined to be a tuple of these previously defined process parameters.

**Definition 11** ***(Process Modes)***
*A process mode $m_p$ of a process $p \in P$ is a tuple of an input data rate interval $r_{in}$ for each of its input channels $in \in Inputs(p)$, an output data rate interval $s_{out}$ for each of its output channels $out \in Outputs(p)$, a latency time interval $lat_p$, a communication region $cr_c$ for each connected channel $c \in (Inputs_p \cup Outputs_p)$, a static $mem_{stat,p}$ and a dynamic memory interval $mem_{dyn,p}$, and a power consumption interval $pow_p$.*

*Associated with each process $p \in P$, there is a non-empty, finite mode set $M_p = \{m_{p,1}, \ldots, m_{p,n_p}\}$ where $n_p \in \mathbb{N}$ is the total number of modes of process $p$.* □

Following from this definition, all previously defined process parameters are no longer directly associated with the process but with the process mode. This expresses the mode-dependency of these parameters. The only exception is the virtuality parameter that remains directly associated with the process as virtuality is not mode-dependent since a process having both virtual and non-virtual modes does not make sense. In examples, often not all parameters of a mode are being specified for brevity and clarity.

A process mode represents a subset of the possible process behaviors, i. e. a mode can be obtained for a subset of process execution paths just like the single interval process behavior is obtained for the set of all possible execution paths of a process. Thus, the single interval representation is equivalent to the representation of a process having only a single process mode.

Coming back to the above examples, the representation of the process of Figure 4.5 using two process modes is

$$m_i = (r_{Q_{in}}, s_{Q_{out}}, lat)$$
$$m_1 = (1, 1, [50, 63])$$
$$m_2 = (1, 0, [14, 21])$$

with mode $m_1$ describing the process behavior for a packet addressed to this receiver, i. e. the execution path including the node *DoLotsOfStuff* and the `write()` function, and mode $m_2$ representing the address miss case, i. e. the execution path including the node *DoAlmostNothing*.

The process of Figure 4.6 can serve as an example for a trade-off between modeling accuracy and problem size. Besides the two extreme cases of representing all execution paths by a single mode

$$m_i = (r_{in}, s_{out})$$
$$m_{single} = ([1, 4], [1, 4])$$

and representing each execution path by a mode of its own

$$m_1 = (1, 1)$$
$$m_2 = (3, 3)$$
$$m_3 = (4, 4)$$

there is also the possibility to merge two of the execution paths and model the process using two modes. For example, if modes $m_2$ and $m_3$ are seen as sufficiently similar to allow efficient analysis, the process can also be represented by

$$m_1 = (1, 1)$$

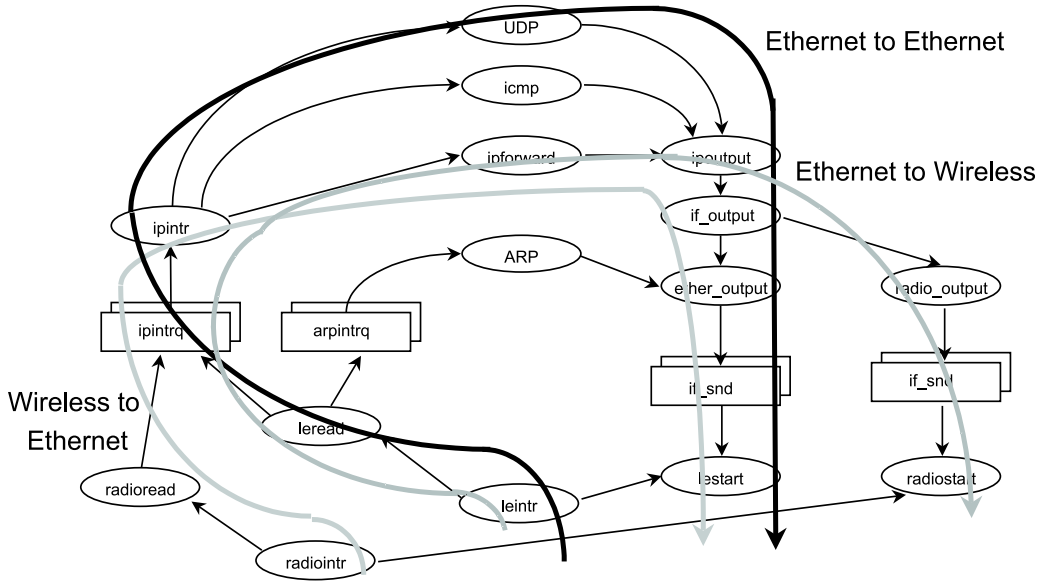



Figure 4.7: Context-dependent flow of execution in a pico-cellular base station

$$m_{23} = ([3,4],[3,4])$$

resulting in a lower modeling accuracy but also a smaller problem size for analysis since the number of modes is reduced. In all cases, the latency parameter is not shown as no information on latency was given in the example.

So far, different execution behaviors of single processes have been distinguished. In the following, it is shown how different execution behaviors of systems or system parts, represented by process networks, can be modeled. Consider the simplified process network implementing the wireless IP standard on a pico-cellular base station as depicted in Figure 4.7. Different base stations of a network are connected by an Ethernet backbone. The example is taken from [96] and was investigated in the ESPRIT MEDIA project.

The bold arrows in Figure 4.7 visualize different execution paths through the network depending on the context of the arriving data packets (e. g. packet from wireless to Ethernet). Depending on this context, processes show different behaviors e. g. process *if_output* sends a packet either to process *ether_output* or *radio_output* depending on the target information in the packet header. Thus in general, a certain execution behavior of a process network is basically the correlation of certain process execution behaviors.

In order to find out how process behaviors are correlated, it has to be determined how process behaviors are chosen. The selection of an execution behavior depends on the execution context of the process. An execution context is defined by constraining the value range of a single or several variables or by introducing relations between variables. By defining a value or value range of these variables, process control structures are determined and an execution path and thus an execution behavior is chosen. An example is shown in Figure 4.8 where the behavior of process $P_1$ depends on the values of two read input tokens. The three possible execution paths (leading to different process behaviors) and their corresponding execution contexts (value definitions) are shown by the bold arrows.

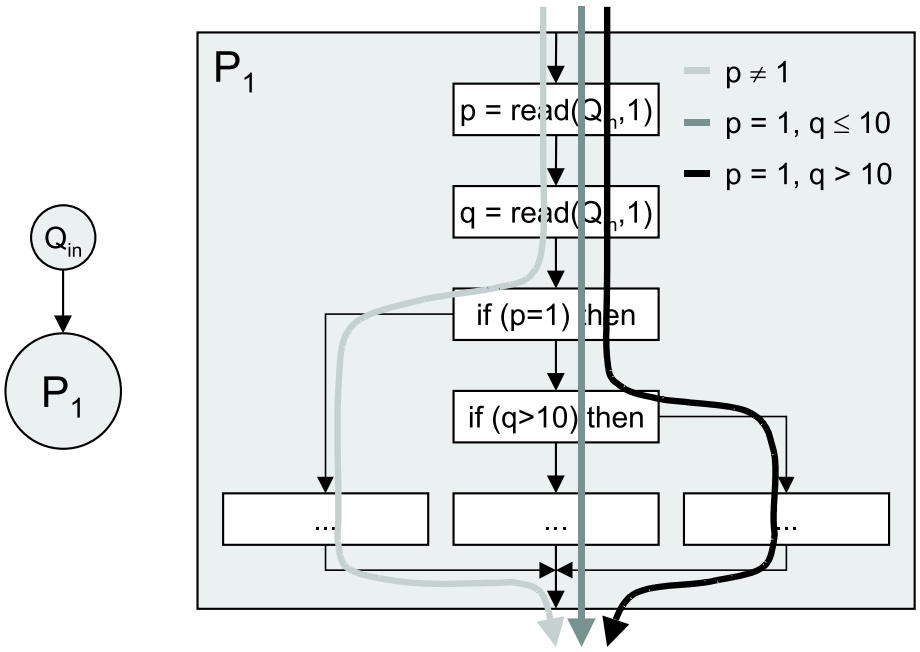



Figure 4.8: Control flow graph with input data-dependent control structures

The execution context of a process is given by its input data and furthermore by its local data denoting a possible state-dependent behavior. The local data of a process can be seen as input data coming from the previous execution of the same process and thus can be modeled by a virtual feedback queue (for examples see Section 6.1.4 on state-based modeling). This queue is virtual as it does not have to be implemented since the local process data is already accounted for by the process parameter $mem_{p,stat}$ (see Section 4.3.2.3). Thus in the following, local process data is generalized and treated as input data as well.

In summary, execution behaviors of processes are correlated by the values of communicated data. In the SPI model, the execution behaviors are modeled using the concept of process modes. The correlation between modes of different processes, however, can not be represented with the previously described set of SPI constructs. This is due to the abstract representation of communicated data as tokens carrying no information on data content or value. While for the majority of data this abstraction still makes sense, the data values relevant for correlation of execution behaviors and thus process modes need to be represented in the SPI model.

In the base station example (Figure 4.7), the correlation relevant data is the packet header as the behavior of the process network depends on the source and the destination of a packet to be processed. But instead of representing the complete packet header or the source and destination addresses in SPI, the abstract information of these data values is represented. For the packets, these are the meaningful combinations of source and destination classes (equivalent to certain address ranges): wireless to Ethernet, Ethernet to Ethernet, and Ethernet to wireless. This abstract information is represented in the SPI model by means of *mode tags* which are associated with data tokens.

**Definition 12** ***(Mode Tags)***
*A mode tag* $(name, val)$ *is a pair of a unique identifier* $name$ *and a value* $val \in D(name)$

*with $D(name)$ being the finite value domain of $name$.* □

Then, e. g. $(header,'w2E')$ denotes that the mode tag $header$ has the value $'w2E'$ which abstracts the information that the data token, the tag is associated to, models a packet which is communicated from a wireless source to an Ethernet destination.

Mode tags are defined to have a finite domain in order to have an enumerable state space for a single token. This is favorable for formal analysis of process correlations. This is no degradation of modeling possibilities as processes also have a finite amount of modes. Furthermore, a single tag value does not have to correspond to a single value for a communicated variable. For example, consider a system with four types of packets (a, b, c, d) that can be classified in control (b) or data packets (a, c, d). Then a mode tag $(type,'data')$ (with $D(type) = \{'control','data'\}$) associated to a data token denotes that this token is a data packet and thus of either type a, c, or d. Similarly, a mode tag representing information for a control structure that compares data values to a threshold typically may have only a binary value set containing of 'greater than threshold' and 'less or equal than threshold' since this is all the information needed for determining this control structure. In the example of Figure 4.8, this may lead to the specification of a tag $q$ with a domain $D(q) = \{'le10','g10'\}$.

Although a tag may have only a single value at a time, it is possible to specify a set of possible values for a tag denoting uncertainty of the current value. This corresponds to the use of intervals for parameters such as data rates. Furthermore, data tokens may have several mode tags denoting different information on the token content associated with them. Both points are accounted for in the definition of a *mode tag set*. In this sense, the following definition extends the mode tag definition.

**Definition 13** ***(Mode Tag Set)***
*Each token $a$ has a mode tag set $TS_a = \{(t_1, V_1), \ldots, (t_n, V_n)\}$ where $t_i \in T$ is a mode tag identifier with $T$ being the global set of such identifiers and $V_i \subset D(t_i)$ is the set of possible values for mode tag $t_i$ with $D(t_i)$ being the finite value domain of tag $t_i$. The mode tag set $TS$ is formulated such that $\forall t \in T : |\{(x, y) \in TS | x = t\}| \leq 1$ denoting that there is at most one tuple regarding each tag identifier contained in a tag set.* □

Unless explicitly specified, the mode tag set of a token $a$ is the empty set. This is interpreted as having no information on the content or values of token $a$. Analogously, the interpretation of a tag set $TS$ that does not contain a mode tag $t$ is that $TS$ does not contain any information on tag $t$. This is equivalent to $(t, D(t)) \in TS$. This interpretation is coherent to the basic principle of the SPI model that the default is uncertainty which can be limited using the SPI constructs.

Using the defined notation of a mode tag set, it is possible to specify uncertainty in terms of several possible values for a single tag (e. g. $(state1, \{'true','false'\})$) but not a possible mutual exclusion for values of different tags (e. g. `either` $(state1, \{'true'\})$ `or` $(state2, \{'true'\})$). However, mutual exclusion of context information can be represented in a natural way by using the same mode tag for the mutually exclusive information (e. g. $(state, \{state1, state2\})$).

Further note that mode tags are a virtual concept, i. e. they do not have to be implemented since they only represent content of data that is already communicated. Thus, no implementation overhead is caused by the concept of mode tags.

In the following, constructs for the generation and evaluation of mode tags will be introduced. Since the specification of content information is only meaningful in correlation to the data containing the content, mode tag production has to be directly coupled with the generation of data tokens. In the SPI model, data tokens are either produced by a process (output data rate) or are generated during channel initialization.

The generation of mode tags for tokens produced during channel initialization can be done straight-forwardly by extending the corresponding Definition 3 to include the possibility to specify mode tag sets for the initial tokens as follows:

**Definition 3** ***(Extended Channel Initialization)***
*Associated with each channel $c \in C$, there is the parameter $d_c = [d_{c,min}, d_{c,max}]$ denoting the initial number of data tokens on channel $c$. Associated with each initial token $a$, there is a mode tag set $TS_a$ with $a \in \mathbb{N}^+$ and $a \leq d_{c,max}$ denoting information on the content of the corresponding token.* □

Assuming an example channel $q$ with an initial number of tokens $d_q = [2, 3]$, the mode tag sets of the initial tokens are specified as follows:

$$TS_1 = \{(u, \{'false'\}), (v, \{4, 5\})\}$$
$$TS_3 = \{(u, \{'true'\}), (v, \{0\})\}$$

This denotes that the first and third initial token have a tag set containing two mode tags each while the tag set of the second token is the empty set. However, the presence of the third token and its tags is uncertain as the initial number of tokens $d_q$ may be 2 or 3. If the initial number of tokens turns out to be 2 the tag set $TS_3$ becomes void.

The production of mode tags by processes is modeled by *mode tag production rules* that are associated with the processes producing the corresponding data. Mode tag production rules depend on input predicates that are defined separately since they will also be used independently.

**Definition 14** ***(Input Predicate)***
*An input predicate is defined on the numbers of available tokens $in.num$ on input channels $in \in Inputs(p)$ of process $p$ and on the mode tag sets $in.tag(k)$ of the tokens read by process $p$ in its current execution where $in.tag(k)$ with $k \in \mathbb{N}^+$ denotes the tag set of the $k$th token in channel $in$. The value of the predicate is either 'true' or 'false'.* □

An input predicate is formulated by a Boolean combination of terms regarding the number of available tokens (e. g. $Q_{in}.num \geq thr$) and terms regarding the values of certain tags (e. g. $('answer', \{42\}) \in Q_{in}.tag(1)$). The restriction for input predicates to only consider mode tags of tokens (to be) read during the current process execution is due to the fact that SPI processes are not allowed to test for input data without reading it (see Section 4.2). As input predicates will be used to select certain process behaviors (mode tag production rules and process modes), this reflects the fact that the behavior of SPI processes only depends on read input data (including its state). This will be discussed in more detail in Section 4.6.4.

The notation of $c.num$ and $c.tag(k)$ in the above definition instead of the usual SPI notation $num_c$ and $TS_c(k)$ reflects the fact that these are not typical SPI modeling parameters but rather variables describing a certain execution situation of a system modeled by a SPI graph.

**Definition 15** ***(Production of Mode Tags)***
*Associated with each mode $m_p$ of a process $p \in P$, there is a finite set of mode tag production rules $TP_p$. Each tag production rule $tp \in TP_p$ is a mapping $tp : I_p \mapsto O_p$ where $I_p$ is the set of input predicates and $O_p$ is the set of output tag production patterns.*

*Each input predicate $i \in I_p$ maps to an output tag production pattern $o \in O_p$ that is activated if the value of $i$ is 'true'. An activated tag production pattern associates mode tags with certain tokens produced on the output channels of the process. The tag set $out.tag(k)$ of the $k$th token produced on channel $out$ during the current process execution is the union of all mode tags associated with this token by activated tag production patterns.* □

As the production of mode tags is directly coupled to the production of data tokens, mode tag production rules have to be mode-dependent just as data rates. Thus, the definition of process modes (Definition 11) is extended to include mode tag production rules.

**Definition 11** ***(Extended Process Mode)***
*A process mode $m_p$ of a process $p \in P$ is a tuple of an input data rate interval $r_{in}$ for each of its input channels $in \in Inputs(p)$, an output data rate interval $s_{out}$ for each of its output channels $out \in Outputs(p)$, a set of mode tag production rules $TP_p$, a latency time interval $lat_p$, a communication region $cr_c$ for each connected channel $c \in (Inputs_p \cup Outputs_p)$, a static $mem_{stat,p}$ and a dynamic memory interval $mem_{dyn,p}$, and a power consumption interval $pow_p$.* □

The output tag production is described by a function $def(c.tag(k), name, value)$ that causes a mode tag $(name, value)$ to be added to the mode tag set of the $k$th token produced on channel $c$. It is possible to specify a set of possible values for the $value$ parameter denoting uncertainty about the value of the produced tag.

For example, consider a process $P_s$ producing packets containing either text data or a frame of a video stream. The type of the produced packet is modeled using two tags, $p\_type$ with $D(p\_type) = \{'text', 'video'\}$ denoting if the packet contains either text or video data and, for video packets, $fr\_type$ with $D(fr\_type) = \{'I', 'P', 'B'\}$ denoting the type of the transmitted video frame. The production of the mode tag $(p\_type, \{'video'\})$ is performed by the following tag production rule

$$tp_v : 'true' \mapsto def(Q_{tv}.tag(1), p\_type, \{'video'\})$$

where the input predicate $'true'$ denotes that the mode tag is produced without the satisfaction of any conditions and $tp_v$ is a label used to reference this rule. Assuming that each of the resulting four different packet types is produced by a separate mode of process $P_s$, the production of the mode tags can be modeled as follows:

$$m = (\ldots, s_{Q_{tv}}, \ldots, TP_p)$$

$$m_1 = (\ldots, 1, \ldots, \{'true' \mapsto def(Q_{tv}.tag(1), p\_type, \{'text'\})\})$$

$$m_2 = (\ldots, 1, \ldots, \{tp_v, 'true' \mapsto def(Q_{tv}.tag(1), fr\_type, \{'I'\})\})$$

$$m_3 = (\ldots, 1, \ldots, \{tp_v, 'true' \mapsto def(Q_{tv}.tag(1), fr\_type, \{'P'\})\})$$

$$m_4 = (\ldots, 1, \ldots, \{tp_v, 'true' \mapsto def(Q_{tv}.tag(1), fr\_type, \{'B'\})\})$$

Then, e. g., process mode $m_2$ produces a token representing a packet containing an I-video-frame on channel $Q_{tv}$ and its mode tag production rules accordingly associate the mode tag set $\{(p\_type, \{'video'\}), (fr\_type, \{'I'\})\}$ to it.

In the previous example, the input predicates have been set to $'true'$ denoting that there was no dependency of the mode tag production rules on the number or mode tags of read tokens. In order to understand why there can be such dependencies and why they need to be modeled, the scope or life-time of tokens and mode tags is examined in the following. The scope of a token and its associated mode tags is a channel. They are produced by an initialization statement or the sending process and are consumed by the receiving process of that channel. Strictly, no token is transfered from one channel to another channel. Rather, the number of tokens read from and produced on a process' input and output channels is correlated by the input and output data rates of the process. The only correlation between certain tokens on different channels can be that they are consumed or produced by the same process execution.[1]

The above correlation is sufficient if data is characterized by its amount. However, in order to be able to model the passing of data content information through a system, means have to be provided to transfer mode tags from one token to a token on a different channel. The following example shows how this can be achieved by mode tag production rules depending on input predicates.

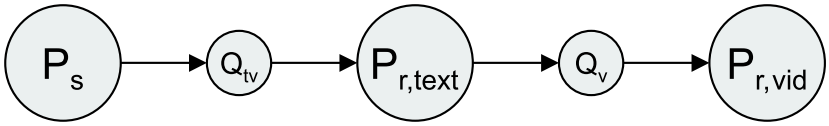



Figure 4.9: Packet transmission system consisting of packet sender $P_s$ and a two-stage receiver ($P_{r,text}$ and $P_{r,vid}$)

Figure 4.9 shows the packet sender of the above example (process $P_s$) and a receiver that is modeled using two processes ($P_{r,text}$ and $P_{r,vid}$). Process $P_{r,text}$ receives all packets but only processes and outputs the text packets while it forwards the video packets to process $P_{r,vid}$ which is responsible for the processing of these packets. Thus, process $P_{r,text}$ can be modeled having the following two modes:

$$m_i = (r_{Q_{tv}}, s_{Q_v}, \ldots, TP_p)$$

$$m_1 = (1, 0, \ldots, \{\})$$

$$m_2 = (1, 1, \ldots, \{tp_1, tp_2, tp_3\})$$

$$tp_1 : (fr\_type, \{'I'\}) \in Q_{tv}.tag(1) \mapsto def(Q_v.tag(1), fr\_type, \{'I'\})$$

$$tp_2 : (fr\_type, \{'P'\}) \in Q_{tv}.tag(1) \mapsto def(Q_v.tag(1), fr\_type, \{'P'\})$$

$$tp_3 : (fr\_type, \{'B'\}) \in Q_{tv}.tag(1) \mapsto def(Q_v.tag(1), fr\_type, \{'B'\})$$

Then, mode $m_1$ models the handling of text packets while mode $m_2$ models the forwarding of video packets to process $P_{r,vid}$. The interesting part of the second mode are the three

[1] This fact is later used to establish a notion of causal dependency between process executions.

tag production rules modeling the transfer of the tag $fr\_type$ denoting the type of the transmitted video frame. In particular, rule $tp_1$ causes a mode tag $(fr\_type, \{'I'\})$ to be added to the tag set of the produced token on channel $Q_v$ if the tag set of the first token on channel $Q_{tv}$ contains the tag $(fr\_type, \{'I'\})$.

This shows how the transfer of mode tags can be modeled using mode tag production rules. In the above example, the tag to be transfered has only three different values such that the tag transfer can be accomplished using only three rules. However, for a mode tag with a larger domain the specification of tag transfer becomes tedious and error-prone. Thus, a function $copy(TS_1, TS_2, t)$ can be defined that copies the information on mode tag $t$ from tag set $TS_1$ to tag set $TS_2$. The definition of this function does not add functionality to the model as it can be replaced by a finite set of rules based on the $def$ function. This set can be generated by $\forall x \in D(t) : (t, \{x\}) \in TS_1 \mapsto def(TS_2, t, \{x\})$ and is finite since $D(t)$ is finite. Using the *copy* function, the description of mode $m_2$ of process $P_{r,text}$ can be simplified to:

$$m_2 = (1, 1, \ldots, \{copy(Q_{tv}.tag(1), Q_v.tag(1), fr\_type)\})$$

The transfer of mode tags can be seen as a special case (identity operation) of a function on tag values. Similarly, arbitrary functions like multiplication or Boolean AND can be defined for mode tags and used in tag production rules. For example, a function $neg(TS_1, TS_2, t)$ modeling the addition of the Boolean inverse of the tag $t$ from $TS_1$ to $TS_2$ can be represented by

$$(t, \{'true'\}) \in TS_1 \mapsto def(TS_2, t, \{'false'\})$$

$$(t, \{'false'\}) \in TS_1 \mapsto def(TS_2, t, \{'true'\})$$

Again, these functions do not add functionality to the model but rather add modeling comfort since they can be represented by a finite set of rules based on $def$ functions.

Besides being evaluated in mode tag production rules, mode tags are utilized to correlate execution behaviors of communicating processes which is why mode tags have been introduced. The adaptation of execution behaviors is equivalent to the selection of process modes that is part of the activation function which is described in the following section.

## 4.5 Process Activation

A SPI process executes without functional interruptions, i. e. during execution a process will not wait e. g. for input data to become available. However, the read accesses of processes to queues are destructive in the sense of tokens being removed by reading processes. Thus, a process may only execute if its required input data is present so that a destructive read access for non-available data can not occur. This is accounted for by basing the SPI process activation on data availability, i. e. a process is activated if its required input data is present and *may* execute if it is activated.

This activation principle is enforced by a *process activation function* which determines on the one hand conditions under which a process is activated, i. e. may execute, and on the other hand possible modes the process may execute in. The selection of possible modes is

a virtual concept as the actual mode selection is distributed in the control structures of the process and performed solely by the process. The activation itself however is substantial to a correct model execution and has to be implemented in terms of a static schedule or a dynamic scheduler.

**Definition 16** ***(Activation Function)***
*Associated with each process $p \in P$, there is an activation function that is formulated as a finite set of rules $\sigma : I_p \mapsto M_p$ where each rule is a mapping of an input predicate $i_\sigma \in I_p$ (see Definition 14) to a finite set of modes $M_\sigma \subseteq M_p$.*

*The activation set $M_{p,a} = \bigcup_{\{\sigma | i_\sigma = 'true'\}} M_\sigma$ is the union of all sets $M_\sigma$ with the corresponding predicate $i_\sigma$ being $'true'$. This set $M_{p,a}$ denotes the set of modes process $p$ may execute in. If and only if $M_{p,a}$ is not the empty set, process $p$ is activated.* □

If the activation function results in an activation set $M_{p,a}$ containing more than one possible mode for an execution of process $p$, it is uncertain in which of the activated modes $m \in M_{p,a}$ the process will execute. The process $P_{r,text}$ of Figure 4.9 can serve as an example for different activation functions. Consider an activation function consisting of the following rule:

$$(Q_{tv} \geq 1) \mapsto \{m_1, m_2\}$$

Then, process $P_{r,text}$ is activated if there is at least one token on $Q_{tv}$. However, it is uncertain in which of its two modes process $P_{r,text}$ will execute as both modes are element of $M_{P_{r,text},a}$ if the process is activated. This is equivalent to the case that input predicates of two or more activation rules match and result in two or more modes being contained in $M_{P_{r,text},a}$. Both cases are legal, and analysis needs to consider that the process may execute in either one of the activated process modes[2]. The uncertainty of the process mode to be executed can be removed using the following activation function

$$(Q_{tv} \geq 1) \wedge (p\_type, \{'text'\}) \in Q_{tv}.tag(1) \mapsto \{m_1\}$$

$$(Q_{tv} \geq 1) \wedge (p\_type, \{'video'\}) \in Q_{tv}.tag(1) \mapsto \{m_2\}$$

where the first rule activates the text handling mode $m_1$ and the second rule activates the video forwarding mode $m_2$.

In examples the activation function is often omitted for brevity. In these cases, the default activation is that each mode is activated (i. e. the process may execute in it) if there are enough input tokens available for an execution.

Evidently, the specified activation function is capable of representing dataflow or event driven models of computation. In Section 6.1.2, it will be shown how time driven activation can be modeled using the SPI activation function.

The activation rules in its general form allow the specification of activation conditions that violate basic principles of the SPI model such as processes executing without functional interruption. Thus in Section 4.6.4, some restrictions for activation rules are defined. These restrictions do not impose additional constraints on valid SPI process behavior. Rather, they enforce process behavior which complies with the SPI execution model.

[2] In general, this is not equivalent to merging both modes parameter by parameter to a single mode as, e. g., shown for the process in Figure 4.6 where a representation with two modes excludes the possible data rate value 2 whereas a single mode representation does not.

## 4.6 Execution Model

While the basic principles of the SPI execution model have already been introduced in Section 4.2, this section provides a more detailed view of the execution model. Furthermore, some of the key decisions regarding the SPI model concepts are justified and restrictions on SPI constructs needed to obtain certain model properties are introduced.

### 4.6.1 Requirements

As the SPI model is intended to be an abstract system representation targeted to analysis and optimization, a key question for the design of such an analytical model is what kind of relationship exists between specification, model, and implementation with respect to the system behavior. In the context of the SPI model which is concerned with non-functional system properties, the system behavior includes the activation, communication, and execution of processes as well as their respective timing. Thus, the following requirements are imposed on the analytical model

- **by the specification:** The analytical model has to be able to represent the behavioral aspects of the different languages used for system specification in a way that the non-determinism with respect to behavioral aspects (e. g. timing or execution order of processes) inherent in the specification is mostly preserved. This is done to keep as much freedom as possible for the system optimization (design space exploration). However, the set of possible behaviors represented by the analytical model may not include system behaviors that are not part of the specification as an implementation based on this model may not be consistent with the specification. In cases where the specified system behavior is not implementable (e. g. zero latency computation), this has to hold for standard implementation interpretations of the specification (e. g. computation latency ”sufficiently” short compared to external system timing).

- **by the implementation:** The analytical model has to conservatively cover all possible behaviors of an implementation as otherwise the analysis results are no longer valid for the implementation. Conservative coverage in this context means that the set of possible implementation behaviors is completely included in the set of possible model behaviors. For example, the latency interval of a process has to include all possible core execution times of the process implementation in order to be conservative. While a conservative latency interval may lead to false rejection of an implementation (as it includes core execution time values that are never actually reached by an implementation), it never leads to false acceptance of an implementation which could result in system failure or violation of a constraint.

  Furthermore, the analytical model should not be overly constraining for an implementation as this may lead to a degradation of the achievable performance of an implementation. An example is an inflexible or inefficient communication behavior of processes in the analytical model which has then to be implemented in order to maintain the consistency between model and implementation. Vice-versa, for the integration of reused components, an overly constraining analytical model may lead

to the inability to cover reused components relying on features not covered by the analytical model.

In summary, not only the specification but also the implementation imposes requirements on an internal design representation targeted to analysis and optimization. The flexibility of this design representation is an important factor for the achievable design quality.

### 4.6.2 Process Communication Timing

This section focuses on different possibilities to model the communication timing of processes. As the main goal of the SPI model is to be able to capture several different input languages including implementation language code (in order to incorporate reused components and legacy code in the design flow), it has to be considered that communication may be requested at any point during process execution. In the following several possible communication timing principles are explored and their suitability is discussed.

A standard communication timing principle is the *atomic buffer update* (ABU) as used e. g. in Ptolemy II [22]. The key advantage of the ABU is its simplicity. A process copies its inputs to internal memory at a defined time (typically process start), stores its outputs internally until it updates input and output channels in a single atomic step at process completion. While this ensures that the termination of a process execution is not going to result in undefined channel states, it results in implementation overhead like the doubled input queue accesses (separate copy and remove), the delay of already generated output data resulting in an artificial delay of dependent processes, and the additional amount of internal memory.

Another standard communication timing used e. g. in task graphs [24] is to have processes read and also remove input tokens at process start and write output tokens at process completion. Like the ABU, this results in defined process communication points but saves the second input queue access. However, the artificial delay of output data and the additional amount of internal memory remain.

One possible remedy for those two problems is to raise the communication timing resolution of a SPI representation by splitting SPI processes at their channel accesses. Clearly, then the generated output data is instantly communicated and no additional memory is required. However besides increasing granularity, the split also affects the behavior. While the SPI subprocesses in the model can be invoked separately, the equivalent process in the implementation is still executed in one piece. This inconsistency between model and implementation behavior can be removed by either splitting up the implementation process as well which may not be trivial e. g. in the presence of loops or introducing a SPI construct which indicates that the SPI subprocesses have to be executed in one piece.

Another remedy for the above problems that allows to increase the communication timing resolution while keeping the process granularity is the concept of communication regions. A communication region denotes for each input or output channel of a process a time interval during which all communication on that channel occurs. This time interval is specified relatively to the process start such that a communication region of $[0, 4]$ means that the communication takes place during the first 4 time units of the process execution. The spreading of communication over the entire process execution instead of having just one or two defined points in time slightly increases the analysis complexity. However, the

apparent problem that communication regions enable writing some outputs before reading some inputs which may lead to a functional interrupt (if a process has to wait for a data token that depends on a data token that was written during the same process execution) and thus to the need for a complicated analysis e. g. based on message sequence charts [50] in order to avoid the generation of deadlocks does not have to be considered. This is due to the SPI activation principle that does not allow the execution of a process without all of its required input data being present.

The decision to choose communication regions as communication timing concept for the SPI model has been based on the following three main arguments.

- Communication regions allow *efficient implementation* as they do not introduce a mandatory overhead as for instance the atomic buffer update. The ABU generally introduces overhead at the modeling level in order to support a certain feature (safe process termination) that may not be needed. Rather, constructs to support such a feature (saving of input and output channels at process start) should be implemented only when needed (process may indeed be terminated).

- Communication regions are *realistic* as real communication does not happen at a point in time but rather during a time interval (barring special operating system constructs). Thus, analysis has to consider a time interval (a communication region) anyway in order to be conservative.

- Communication regions allow *flexible* adaptation of communication timing. For a top-down system design approach, a certain fixed communication timing can be enforced by process wrappers e. g. delaying all outputs until process completion. For analysis of existing implementations or implementations designed using a fixed tool flow, one has to be able to represent implemented communication behavior possibly allowing communication throughout process execution. Thus, the communication timing resolution has to be increased and communication regions are preferable in comparison to the presented process-splitting approach.

The exact definitions concerning communication regions in the SPI model are given in Section 4.3.2.1.

### 4.6.3 Communication Constructs

This section clarifies which basic communication constructs can be represented using the SPI model. In this context, basic constructs are the direct access mechanisms to the SPI channels as defined in Section 4.2. The SPI channels, which are in case of queues one-to-one and in case of registers one-to-many communication links, are accessed asynchronously (i. e. buffered) and allow unidirectional communication. More complex communication mechanisms like rendezvous or client-server communication or communication with not directly supported properties (synchronous, bidirectional) can be represented by a combination of the basic communication constructs and thus can also be modeled in SPI. Examples are shown in Section 6.1.3.

In Section 4.2, channel accesses of processes have been defined to be destructive read and non-destructive write for queues as well as non-destructive read and destructive write

for registers. The remaining communication property not yet specified is blocking behavior meaning the behavior of a communication construct in case of failure to perform the communication request. While due to channel properties register accesses as well as the writing queue access (as queues are unbounded) are always successful and thus non-blocking, the reading queue access depends on the channel state, i. e. it can only be successful if the queue holds at least the number of tokens which the process wants to read. In the following, different possible communication constructs for reading queue access and their behavior in case of a failure are examined.

The basic and most common construct is the blocking read access, here denoted by the `receive()` function. In case a process tries to consume a token from an input queue using this function, the process either successfully consumes the token or blocks if there are not enough tokens on the input queue. In terms of the SPI model, this blocking would violate the SPI concept of processes executing without functional interruption. Thus, the activation function has to assure that a process may only start execution if there are enough tokens on each input queue to prevent an unsuccessful `receive()` access and thus a blocking of the process.

Another communication construct, in the following denoted as `try_receive()` function, differs from the `receive()` function in the way an access failure due to missing data is treated. Instead of blocking the process, the `try_receive()` function returns an error code and continues process execution. Such a function can be used e. g. to sample an input queue for arriving data or to implement a non-determinate merge and is supported by many real-time operation systems. In the SPI model, a `try_receive()` usually results in a data rate interval or two process modes depending on the number of available tokens.

While the `try_receive()` function always consumes data tokens if they are available, one could envision a function that only tests if a certain number of input tokens is available without consuming any. While this function shows no destructive read characteristics, it follows from the two previous functions in the sense of `receive()` always consuming a token, `try_receive()` consuming a token only in case of availability, and the test function never consuming a token. Such a test function however creates new dependencies between processes and hidden sources of non-deterministic behavior but does not seem to add useful functionality. Thus, a test for input data without consuming it is forbidden in the SPI model.

In summary, while register accesses and writing queue accesses are well-defined by channel properties only, SPI supports the representation of two different host language communication constructs (`receive()` and `try_receive()`) for reading queue access.

### 4.6.4 Process Activation

In the SPI model, process activation is based on data availability, i. e. a process may perform computations if the data required to perform these computations is present. This activation principle is widely used, very intuitive and somewhat minimalistic as it only enforces the data dependencies of an application.

A simple implementation of this activation principle are Kahn graphs [58] (see Section 2.2.2.1) which ensure the availability of input data for the constantly enabled pro-

cesses by relying only on blocking read semantics. Then, the blocking mechanism suspends the process if the required data is not available and activates the computations until the next blocking read if the input data becomes available. The downside of the simple implementation is the resulting context switching overhead (e. g. [92]).

A way to avoid this implementation overhead is to raise the computational granularity, i. e. the amount of computations performed per activation. For Kahn graphs, this granularity is typically very low with the extreme case being two consecutive blocking read statements where the first blocking read statement does not activate any computations. For dataflow process networks [69], Kahn processes are divided into *firings* containing several blocking read statements and their depending computation in order to raise the computational granularity. Each firing consumes and produces data tokens and is activated by a firing rule determining when a process may fire.

The SPI model uses the same clustering approach with firings being equivalent to process modes and firing rules being equivalent to the activation function. In this context, the activation function has to ensure the availability of the required input data tokens such that no read access blocks and thus results in a functional interruption of the process which would be a violation of the SPI modeling assumptions. Moreover, it would also contradict the reason why these statement were merged to firings or process modes, respectively.

As SPI processes may have different modes with different data consumption patterns, the number of tokens that have to be available on the input channels for the activation of a process depends on the process mode. Thus, most of the information relevant for mode selection has to be known for process activation, as well, such that, as already described in Section 4.5, the activation function is also used to represent the mode selection resulting in a more compact model as compared to the separate representation of process activation and mode selection.

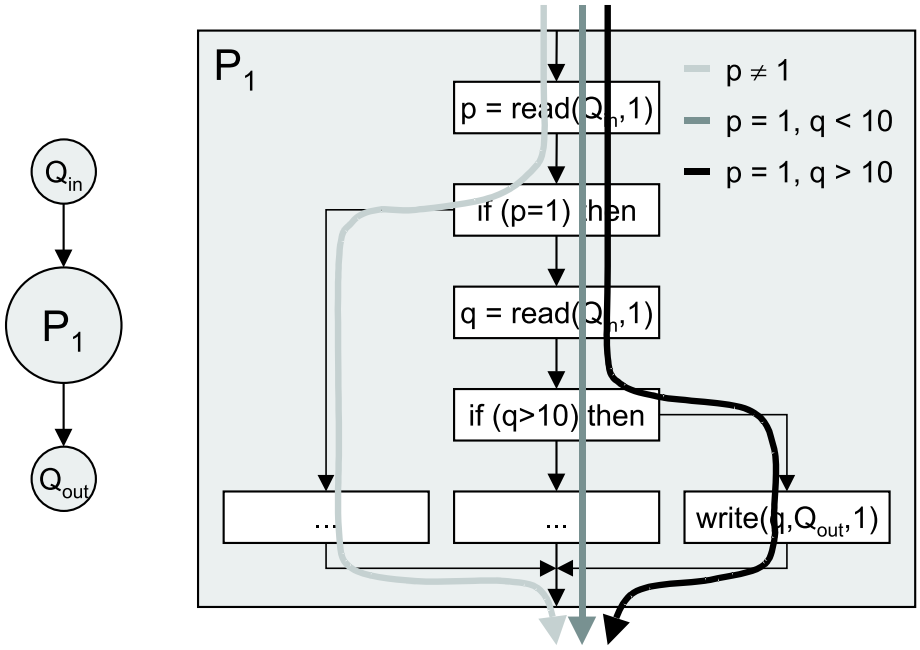



Figure 4.10: Control flow graph with input data-dependent consumption and production behavior

The interplay of process activation and mode selection can be best explained using an example. For instance, the control flow graph depicted in Figure 4.10 has three possible execution paths that can be modeled by three process modes $m_{lg}$ (light gray arrow), $m_g$ (gray arrow), and $m_b$ (black arrow). While modes $m_g$ and $m_b$ consume two tokens and thus can only execute without functional interruption if $Q_{in}$ contains at least two tokens, mode $m_{lg}$ requires only one available token in $Q_{in}$ for a proper execution. However, this does not mean that process $P_1$ may always be activated if one token is available nor does the activation of $P_1$ always require two available tokens. Rather does the number of required tokens depend on the first control structure and thus on the value of the first token consumed from channel $Q_{in}$. This value may be represented by a tag $p$ with a domain $D(p) = \{'one', 'notone'\}$. Then, the activation function for $P_1$ can be formulated as follows:

$$(Q_{in}.num \geq 1) \wedge (p, \{'notone'\}) \in Q_{in}.tag(1) \mapsto \{m_{lg}\}$$

$$(Q_{in}.num \geq 2) \wedge (p, \{'one'\}) \in Q_{in}.tag(1) \mapsto \{m_g, m_b\}$$

Assuming an empty queue $Q_{in}$, the arrival of a token with tag $(p, \{'notone'\})$ will activate process $P_1$ while the arrival of a token with tag $(p, \{'one'\})$ will require a second token to arrive before $P_1$ is activated. In other words, after the arrival of the first token, it is known if $P_1$ will execute in mode $m_{lg}$ or not although the actual determination will not happen before reaching the first control structure of $P_1$.

Finding the relationships between data consumption and control structures has been easy for this simple example. For more complex processes, the mode-dependent number of tokens required for activation can be formally determined by performing a context-dependent path analysis (see Section 6.2.1 for more details).

In summary, the activation function can be seen as forward propagation of process control structures and input queue accesses. At activation time, all required input data is present (otherwise there would be no activation). Thus, all process control structures are determined and the determination is known if it is modeled using mode tags. Thus, the process mode can be determined at activation time as well although the actual selection of the execution path (set), represented by this mode, which is distributed over the process control structures is not yet performed.

An exception to this complete knowledge about process control structures at activation is the `try_receive()` function that is itself essentially an undetermined control structure as at the time of activation it may not be clear if the input tokens `try_receive()` tries to read are available or not. Furthermore, the `try_receive()` function may provide new data to determine additional control structures. A consequence of these control structures being undetermined at activation is that the predicted process mode may change during process execution.

This effect is demonstrated using the example process $P_t$ depicted in Figure 4.11 which has two modes $m_1$ ($r_{Q_{in}} = 1$) and $m_2$ ($r_{Q_{in}} = 2$). The activation function of $P_t$ may be formulated as follows:

$$(Q_{in}.num = 1) \mapsto \{m_1\}$$

$$(Q_{in}.num \geq 2) \mapsto \{m_2\}$$

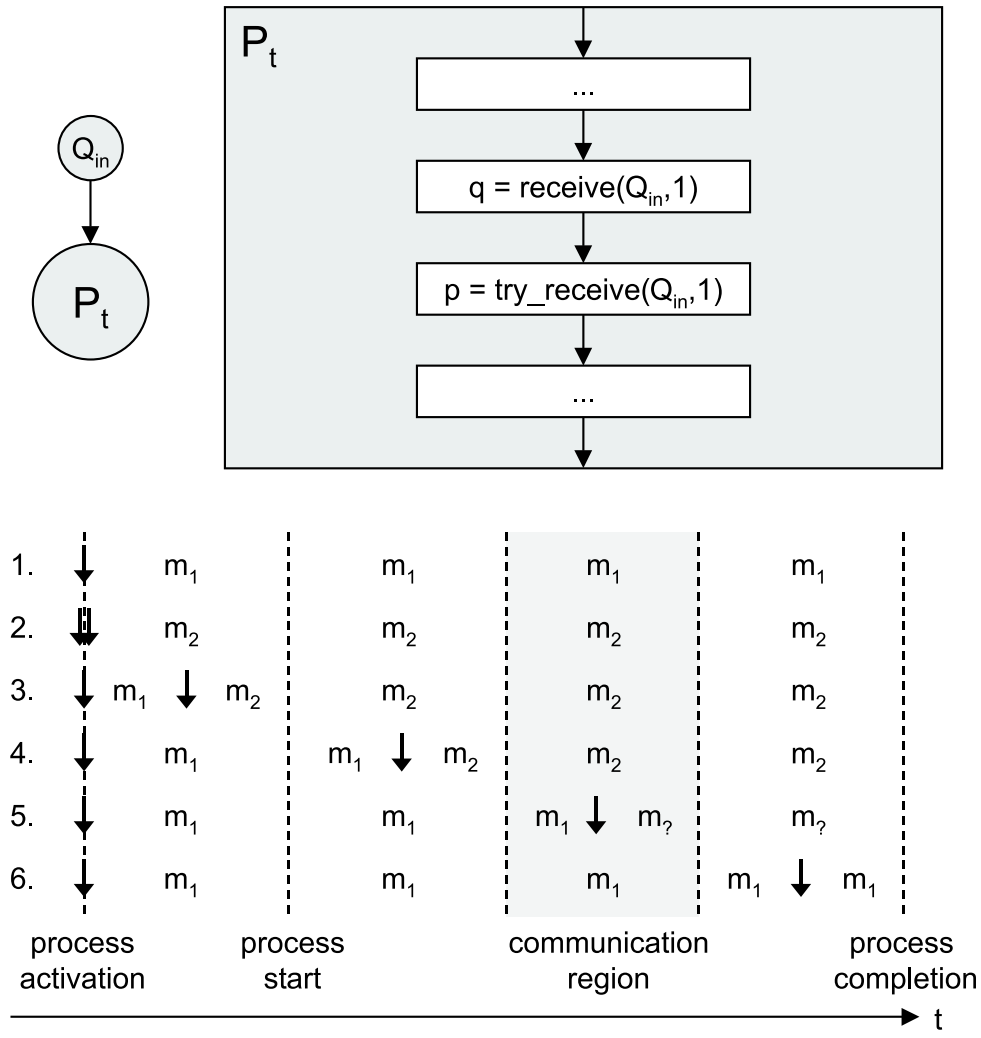



Figure 4.11: Control flow graph with a try_receive statement and different execution scenarios

For different execution scenarios where downward arrows denote the arrival of data tokens, the performed process mode is compared to the predicted process modes at activation and process start. For scenarios 1,2, and 6 all three process modes are the same as no additional tokens arrive between activation and the end of the communication region of $P_t$ so that the `try_receive()` function is not successful. In scenario 3, an additional token arrives before process start leading to a successful `try_receive()` and a change of the predicted process mode between activation and process start while the mode remains constant during execution. If the additional token arrives after process start (scenario 4) the process mode changes during run time. If this arrival occurs during the communication region, the process mode even becomes uncertain as it can not be analyzed if the token arrived before or after the `try_receive()` function is executed. While harmless in this example, the use of the `try_receive()` function is potentially dangerous since it may lead to the invalidation of a process activation as will be shown in a later example.

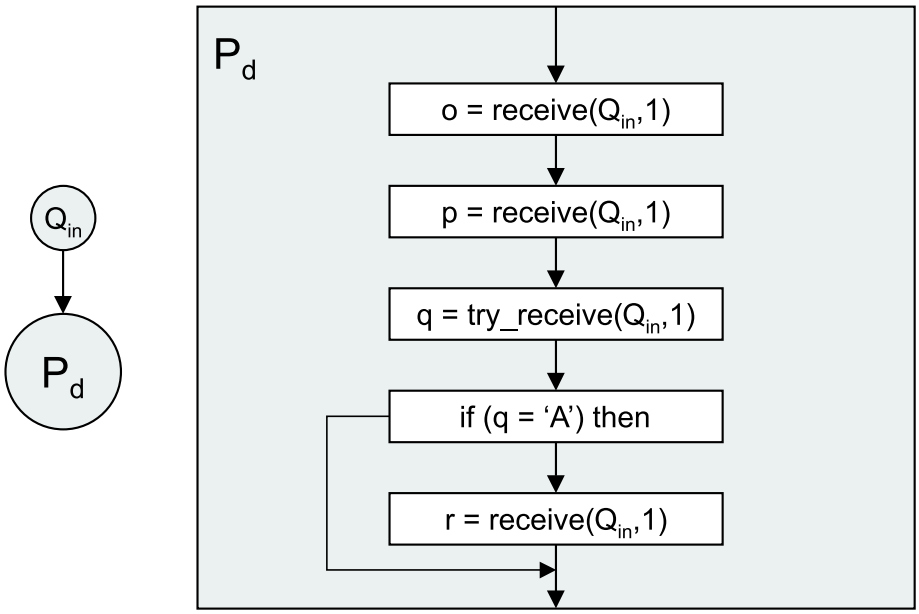



Figure 4.12: Control flow graph of a process demonstrating the possible invalidation of an activation

The SPI activation function in its general form allows the specification of activation conditions that violate basic principles of the SPI model such as the SPI modeling assumption of processes executing without functional interruption. Thus in the following, some restrictions for activation rules are defined to prevent these violations. These restrictions do not impose additional constraints on valid process behavior but rather allow a check for correct modeling.

- **Activate only if enough input data is present.** As defined in Section 4.5, the activation function is formulated as a set of mappings $\sigma$ of an input predicate $i_\sigma$ to a set of process modes $M_\sigma$ which are activated if $i_\sigma$ is 'true'. Then, the predicate of a rule $\sigma$ has to be 'false' if in some input queue $Q_{in}$ there is a smaller number of available tokens than possibly consumed by the process in any mode $m \in M_\sigma$. This can be formally defined as

  $$\forall m \in M_\sigma : (Q_{in}.num \leq r_{Q_{in},max}) \Rightarrow i_\sigma = 'false'$$

  Otherwise, the execution of this process may lead to an attempted consumption of non-available data and the process could be blocked violating the modeling assumption.

- **A process activation may not be invalidated.** The invalidation of an activation means that the content of the input queues has changed such that the input queues do not hold the required input data for any process execution. Assuming a process has been activated and has already started execution, this activation may not be invalidated as otherwise the process would not be able to complete without functional interruption.

An example is the process $P_d$ in Figure 4.12 which has three different modes $m_1$ ($r_{Q_{in}} = 2$), $m_2$ ($r_{Q_{in}} = 3$), and $m_3$ ($r_{Q_{in}} = 4$) which are selected according to the following rules

$$(Q_{in}.num = 2) \mapsto \{m_1\}$$

$$(Q_{in}.num \geq 3) \wedge (val, \{'notA'\}) \in Q_{in}.tag(3) \mapsto \{m_2\}$$

$$(Q_{in}.num \geq 4) \wedge (val, \{'A'\}) \in Q_{in}.tag(3) \mapsto \{m_3\}$$

Assuming the presence of two tokens on channel $Q_{in}$, process $P_d$ is activated and predicted to execute in mode $m_1$. Then, the arrival of a third token with the associated mode tag $(val, \{'A'\})$ on channel $Q_{in}$ after the start of process $P_d$ results in the invalidation of its activation as none of the activation rules matches anymore and in process $P_d$ being blocked due to an unsuccessful blocking read access.

For the activation function follows that processes may not be activated if the state of the input channels may change in a way that results in an invalidation of the activation. For the above example, this means that the activation function has to be changed such that the first rule is deleted. This results in mode $m_1$ never being activated but in a correct activation of process $P_d$ which can not be invalidated.

For the formulation of a valid activation function in general follows that after at least one predicate of an activation rule becomes true (leading to a process activation), at least one predicate has to remain true until process completion. This prohibits e. g. the use of a predicate $(Q.num = x)$ (with $Q$ being the single input queue of a process) if there is no other predicate that is 'true' for $(Q.num > x)$, since the arrival of the $(x + 1)$th token on $Q$ would invalidate the current process activation.

Furthermore, the activation may not depend on the value of mode tags associated with tokens of an input *register* as the register token can be overwritten by the writing token at any time resulting in a possible invalidation of the activation.

After having described the necessary restrictions on the activation function in order to comply with the SPI modeling assumptions, the remaining questions are when the activation function has to be evaluated and how the activation function has to be implemented.

The input predicates of the activation rules and thus the activation function are dependent on the number and tag sets of data tokens on the input channels of a process. Thus, the activation function may change whenever the state of one of the input channels changes.[3] However, since no process activation may be invalidated and possible mode changes are performed by process control structures, the activation function only needs to be evaluated at each input channel change until an activation occurs. This clearly reduces the run-time overhead for an implementation based on dynamic scheduling.

For analysis however, an evaluation of the activation function during process execution or at process completion results in additional information on the executed process mode (see Figure 4.11). This information can be used in order to more narrowly bound the number and content of produced data tokens. Due to the consumption of data tokens, the state of the input queues changes during process execution. In order to prevent that these

[3]However, note that if no `try_receive()`-like construct has been used in the process description the input predicates do not change after process activation.

changes result in changes of the input predicates of the activation function, the decrement of the $c.num$ and $c.tag(k)$ constructs due to token consumption is delayed until process completion.

As the activation function determines *when* a process is ready to be executed, the implementation equivalent of the activation-related part of the activation function is the process scheduler (whereas the mode selection is performed by the process control structures itself). Different approaches to process scheduling are static and dynamic scheduling. For static scheduling, the process execution order is determined and coded in a scheduling table based on an off-line analysis of all process activation functions. In contrast for dynamic scheduling, a scheduler is synthesized from the activation functions that dynamically determines the process execution order at run-time.

A possible problem for the scheduler implementation is a dependency of the process activation on mode tag values. Such a dependency occurs for example when representing Boolean data flow actors [8] where the number of consumed input tokens and thus the actor activation depends on the Boolean value of a token to be consumed from a queue connected to the dedicated control input. This Boolean value can be abstracted by a mode tag in the SPI model to distinguish the different consumption behaviors. Then, the activation of a corresponding SPI process depends on the value of this mode tag. In order to implement a dynamic scheduler considering mode tag dependent process activation, the correspondence between mode tag values used in the activation function (e. g. $(packet\_type, \{'control'\})$) and the 'real' transmitted data (e. g. second bit of packet header distinguishes between control and data packets) has to be available. This can be achieved by providing and using this correspondence (abstraction knowledge) for scheduler synthesis and let the scheduler test for data values in the implementation or by implementing the activation-relevant mode tags as messages to the scheduler.

### 4.6.5 Monotonicity

A useful property for design space exploration for process networks is the independence of the results from the execution order of the processes. This allows the process ordering to be optimized within the partial order imposed by existing data dependencies (queues) in order to minimize e. g. memory or response times. A process network having this property is said to be determinate or *monotonic* [58, 69]. Here, monotonicity means that the sequence of produced outputs grows with the sequence of consumed inputs, i. e. a produced output token will not be invalidated by the consumption of additional data. Examples for monotonic process networks are Kahn or SDF process networks.

Clearly, a SPI process network being translated directly from a monotonic process network retains this property. Due to the intended use of the SPI model as an internal design representation being able to represent different models of communication however, a general SPI process network may not be monotonic as this would prevent the representation of non-monotonic models of computations. The sources of non-monotonic behavior in a SPI process networks are as follows:

1. A process containing a `try_receive()`-like construct may produce different outputs (differing in amount as well as in value) depending on the number of tokens on its input queues. These numbers in turn depend on how often the processes writing to the input queues have already been executed, leading to process execution order

dependent network behavior. Thus, tokens may be produced that would not have been produced if additional input tokens would have been there.

2. The process mode and thus the number and value of produced data tokens may depend on the value (mode tags) of tokens communicated via registers. Register communication is essentially unsynchronized unidirectional shared variable communication. As Lee states in [69], shared variable communication may introduce non-determinism to process networks since the result of a process execution reading a shared variable may depend on whether the process writing the variable has already been executed or not. Some dataflow process network descriptions such as the Navy's processing graph method (PGM) [65] allow processes to share variables. One of the reasons to include register communication in the SPI model was to explicitly visualize the sharing of variables in order to being able to identify this source of non-determinism and non-monotonicity.

3. Furthermore, in the SPI model the host languages and their functionality are not defined. Thus, processes may have additional internal non-determinism that is not recognizable such as in the case of `try_receive()`. However, it is assumed that SPI processes are functional in the sense of Kahn [58] meaning that the output sequences are a function of the input sequences. Thus based on the assumption of correct models, the possibility of unspecified internal non-determinism does not have to be considered.

However, a SPI process network may show monotonic behavior *in the range of the specified determinism*. By restricting monotonicity to the range of specified determinism, sources of non-determinism are not considered as long their influences are not specified and thus can not be identified on the SPI level. An example is a simple unsynchronized register communication between two processes. In an implementation of the reading process, the values of the data it produces will clearly depend on the data read from the register and thus also depend on the execution order of the communicating processes. If on the SPI level however this influence is not specified using mode tags, the non-determinism is not observable and it is assumed that a system implementation is correct regardless of the particular behavior. Thus, a SPI process network containing this communication is considered to be monotonic. If however dependencies on values of register tokens are specified, the SPI network is considered to be not monotonic since such a specified dependency observably influences the network behavior.

Based on this discussion, the following conditions can be formulated to yield monotonic behavior in the above sense:

- On each input channel $in \in Inputs_p$, each process $p \in P$ may never consume more tokens than the least amount needed for activation of process $p$ as denoted by

$$\forall m_p \in M_p : r_{in,max} \leq \min_i(a_{in,i})$$

where $a_{in,i}$ denotes the least number of tokens on channel $in$ required for activation of process $p$ in mode $m_i$ and $i = 1, \ldots, n_p$ with $n_p$ being the number of modes of process $p$. By ensuring that the process is not activated before there are sufficient tokens for all possible process behaviors, essentially `try_receive()` functions

can never be unsuccessful and thus are equivalent to blocking reads which enforce monotonic process behavior [58].

- For each process $p \in P$, the process mode selection has to be independent of mode tags associated with tokens stored in input registers of $p$. This tightens the limitation that process activation may not depend on register tokens (see Section 4.6.4). Furthermore, the mode tags of the produced output tokens have to be independent of the mode tags associated with tokens stored in input registers of $p$. Together, this results in the restriction that input predicates used in the activation function or in tag production rules of process $p$ may not contain any reference to mode tags of tokens stored in an input register $r_{in} \in (Inputs_p \cap R)$. This way, a specified dependency of the behavior of process $p$ on the values of input registers as well as the specified propagation of these dependencies is excluded.

Then, if a SPI process network satisfies these conditions, it is called monotonic in the range of specified determinism and the correctness of the system is considered to be independent of the chosen process execution order. This execution order is only limited by the partial order defined by the token dependencies on the SPI queues. SPI queues may also be used in order to enforce monotonicity by synchronizing register communication. An example for this is the translation of Simulink to SPI as described in Section 6.2.2.

Please note that processes having data rate intervals are not excluded from being part of a 'monotonic in the range of specified determinism' SPI process network. This is valid since assuming functional process behavior the number of consumed and produced data tokens is independent of the number of present input tokens if the network satisfies the above conditions.

## 4.7 Function Variants

Many embedded systems are implemented with a fixed core function and a set of alternative *function variants* to adapt the system to different applications or environments. Examples are TV sets which can be adapted to different standards or automotive control systems used in countries with different emission laws. Function variants are mutually exclusive, i. e. only one variant of a set of alternative functions is selected at a time. There may be several of those variant sets in one embedded system, e. g. for different input and output standards of a multimedia device. The variant selection for these sets may be related or independent.

Function variants can be classified into different types according to the stage of the product life-time during which the selection of the function variant occurs. *Production variants* are selected at production time, e. g. by downloading a certain software variant into an EPROM. *Run-time variants* are selected at system start-up time, e. g. as part of a boot sequence which reads switches or flash memory stored parameters. In both cases, system optimization can assume that the system's variants can not be changed during system operation. A more complicated selection process is found in dynamically *reconfigurable architectures*. Here, a subsystem is typically (re-)configured by a higher level controller during run-time to execute a function which the subsystem itself cannot change.

In [83], the SPI model has been extended to facilitate the representation of function variants and their selection. In the following, the new SPI elements and the basic ideas of function variant representation are introduced. Detailed information on function variant selection can be found in [27, 83].

Clearly, a single SPI process with a set of modes can be used to represent function variants, where each of the variants is mapped to a single mode or a set of modes of that process. Then, the variant selection maps to mode selection inside the process. The drawback of this representation is that the modeling is too coarse grain, since function variants usually incorporate several processes and channels, i. e. whole subgraphs, instead of a single process. To keep the level of granularity, the subgraphs of all function variants can be included in the system model together with some coordination framework that is responsible for the distribution of the data according to the currently selected subgraph (by means of mode tags). In this case, the information about the mutual exclusion of the function variants is distributed within the process network and can hardly be recovered. Thus, the use of processes and process modes is not sufficient for a reasonable representation of function variants.

As already indicated, changing a system's variant in the functional description corresponds to exchanging subgraphs in SPI. Such subgraphs may be represented as *clusters*. A cluster contains a set of graph elements which communicate through the cluster border via input and output ports. This concept allows for hierarchical construction of complex SPI models and enables stepwise refinement.

**Definition 17** ***(Cluster)***
*A cluster is a tuple* $\gamma = (I, O, P, C, E, \Psi)$ *where* $I$ *denotes the set of input ports and* $O$ *the set of output ports, respectively.* $P$ *denotes the set of embedded processes,* $C$ *the embedded channels, and* $E \subseteq ((P \cup \Psi) \times (C \cup O)) \cup ((C \cup I) \times (P \cup \Psi))$ *the embedded edges.* $\Psi$ *denotes the set of embedded interfaces to be defined later. In addition to the constraints on channels as given in Def. 1, an input port* $in \in I$ *satisfies* $indeg(in) = 0$, $outdeg(in) \leq 1$ *and an output port* $out \in O$ *satisfies* $outdeg(out) = 0$, $indeg(out) \leq 1$. □

Clustering does not add functionality to the model and is only a structuring concept. The only restriction in this context is that a cluster, like a process, can only be connected to channels.

Now, a system with two function variants can be represented consisting of three parts. The first common part contains all elements that are not variant-dependent, while the remaining parts are mutually exclusive clusters which represent the distinct function variants. Evidently, both clusters must have the same external connections in terms of input and output ports, since otherwise they could not be reasonably exchanged by each other. In other words, the three parts need a common *interface*. Furthermore, information is needed about a reasonable *mapping* between interface ports and cluster ports.

**Definition 18** ***(Interface)***
*An interface is a tuple* $\psi = (I, O, \Gamma, PortMap)$ *where* $I$ *denotes the set of input ports,* $O$ *the set of output ports,* $\Gamma$ *the set of clusters associated with this interface, and a partial function* $PortMap$ *which maps the input and output ports of each cluster in* $\Gamma$ *to an input*

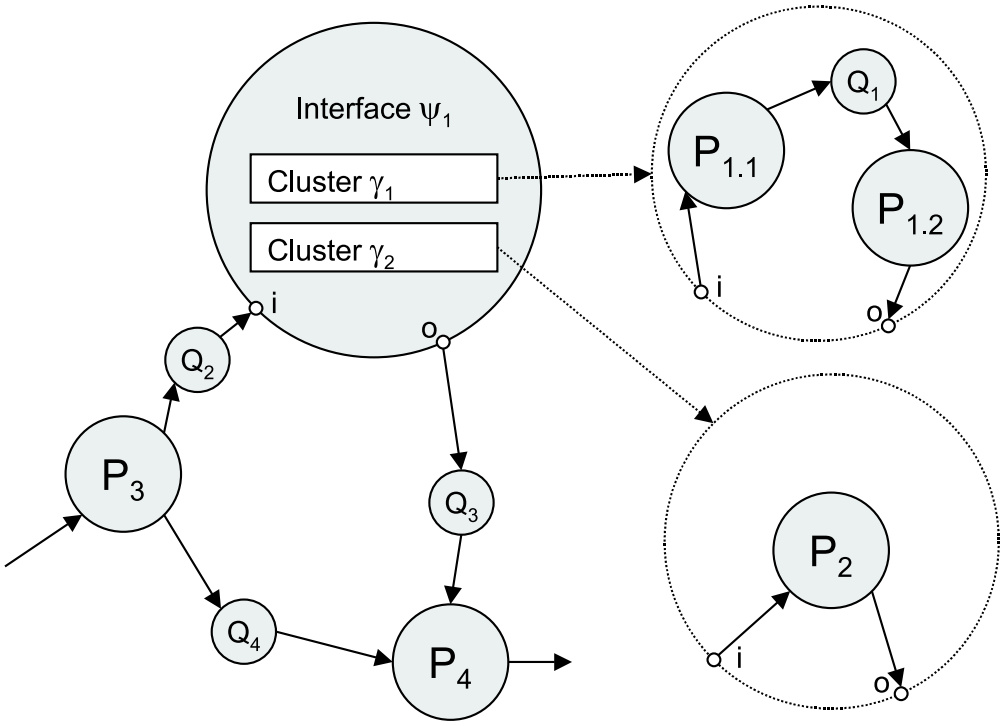



Figure 4.13: Example system with two function variants

*or output port of the interface $\psi$, respectively. If an interface is embedded into a cluster, the indegree of input ports and the outdegree of output ports is at most one.* □

Using these two constructs, a system part for which different function variants exist can be represented by an interface with an associated set of different clusters. Then, each function variant is represented by exactly one of the clusters that can be connected to the interface according to the corresponding mapping in $PortMap$. An example for a system part with two function variants is the interface $\psi_1$ in Figure 4.13. The different variants are depicted as clusters $\gamma_1$ and $\gamma_2$. The port map is not explicitly shown but implicitly represented by a reference-by-name schema, i. e. corresponding cluster and interface ports have the same name.

## 4.8 System Environment Modeling

An essential part of an embedded system specification are the requirements and constraints that are imposed by the environment (”bed”) the system is embedded in. These requirements and constraints limit the design space, provide optimization goals, and may be used to guide the design process. Thus, the design methodology has to regard the system environment. For the SPI model follows that it has to provide means to capture and model the system environment. In this section it is shown how SPI allows to coherently model system and system environment including constraints.

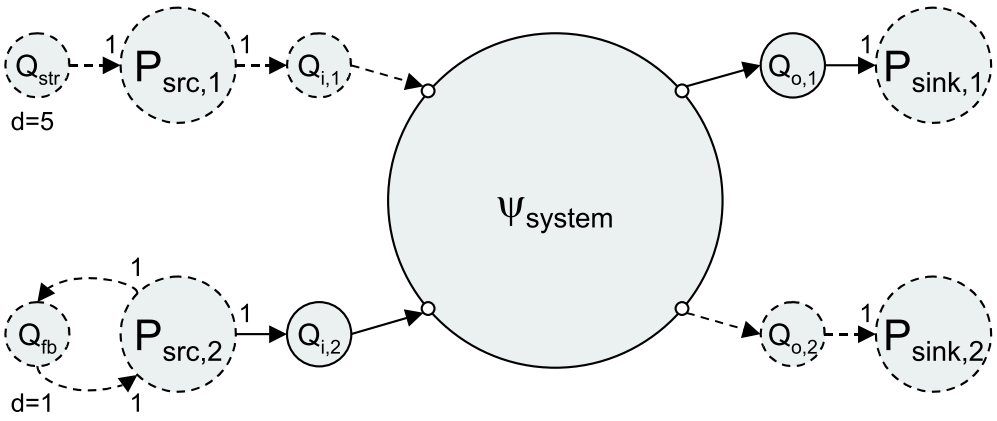



Figure 4.14: SPI graph modeling a system and its environment

### 4.8.1 Virtual Model Elements

One of the uses for virtual model elements is the modeling of the system environment. From the definition of virtuality follows that virtual model elements have the same semantics as non-virtual ones but do not have to be implemented. Both of these characteristics are excellent for the representation of the system environment by virtual model elements. First of all, the system environment is either already implemented or at least the implementation of the environment is not part of the design of the embedded system. And secondly, the fact that the semantics of the model elements are independent of the virtuality flag allows for a seamless interfacing between the system environment (virtual model elements) and the system to be implemented (non-virtual model elements).

In the following, the use of virtual model elements for the modeling of the environment is demonstrated. Figure 4.14 shows a SPI representation of a system denoted by the interface $\psi_{system}$ and its environment consisting of two signal sources ($P_{src,1}$ and $P_{src,2}$) and two signal sinks ($P_{sink,1}$ and $P_{sink,2}$).

Processes $P_{src,2}$ and $P_{sink,1}$ are connected to the system via non-virtual channels denoting that the communication between system and environment has to be implemented. $P_{src,1}$ and $P_{sink,2}$ however are connected to the system via virtual channels. This means that the communication is part of the environment and only a proper interface on the system side has to be implemented.

The signal source modeled by process $P_{src,1}$ produces only a finite amount of data tokens ($5$) for the system. This is modeled by the five preassigned tokens in channel $Q_{str}$ that has only a reading process ($P_{src,1}$) but no process that adds token to it. On the contrary, signal source $P_{src,2}$ produces an infinite number of tokens for the system. This is modeled by the virtual feedback channel $Q_{fb}$ and process $P_{src,2}$ having an input and output data rate of $1$ data token per execution for channel $Q_{fb}$. With one initial data token on the channel ($d_{Q_{fb}} = 1$) supporting the first activation, each execution of $P_{src,2}$ then enables its following activation.

Another modeling possibility not shown in Figure 4.14 is a feedback channel from a signal sink to a signal source denoting a possible dependency of a system input on a system output as typically found in control systems.

After providing means to model the system environment structure, the following sec-

tions show how to model constraints imposed by the environment and properties of the environment.

### 4.8.2 Constraints

Before looking at the representation of constraints in the SPI model, it is important to understand the difference between constraints and properties. A property is typically the result of system implementation and describes a certain aspect of the behavior a system shows (e. g. ”9.3 ms after the crash the airbag is released”). A constraint however is part of the system specification and formulates a certain condition that the behavior of the system has to fulfill in order to satisfy the specification (e. g. ”no later than 10 ms after the crash the airbag must have been released”).

In the SPI model, system properties are described using the parameters as presented in Section 4.3. This section introduces the modeling of non-functional constraints using SPI. Functional constraints (e. g. each set of inputs on channel A has to be sorted in descending order and written to channel B”) as possible to specify in e. g. the declarative Rosetta language [48] are not part of the SPI model as the focus of the SPI model is on the validation and realisation of non-functional system properties and the system function is already specified using the different input languages.

A special focus is set on timing constraints but constraints concerning other properties such as power consumption and are discussed as well.

#### 4.8.2.1 Timing

Depending on application and environment, timing constraints imposed on embedded systems vary widely. In the SPI model, there is a basic type of timing constraints that is used together with virtual model elements to model all other types of timing constraints. This is the *latency path constraint* limiting the time tokens may take to travel along these paths.

**Definition 19** ***(Latency Path Constraints)***
*A path constraint is a labeled path in the SPI graph. A path is of the form*

$$(P_1 \longrightarrow C_1 \longrightarrow P_2 \longrightarrow \ldots \longrightarrow C_n \longrightarrow P_{n+1})$$

*while involving* $n+1$ *processes,* $n$ *channel nodes and* $2n$ *edges. A latency path constraint*

$$LC_{path} = [t_{lat,min}, t_{lat,max}]$$

*restricts the time interval between the time (*$t_u$*) a token* $u$ *is written to the first channel of the path* $C_1$ *and the time (*$t_v$*) a causally dependent token* $v$ *is read from the last channel of the path* $C_n$*. The path constraint must be satisfied for any sequence of causally dependent tokens during any possible execution of the system such that always*

$$t_{lat,min} \leq (t_v - t_u) \leq t_{lat,max}$$

□

The causal dependency between tokens expresses a possible value dependency between them and can be defined recursively as follows. A token $v$ is said to be causally dependent on another token $u$, if $v$ was produced by a process execution which either read $u$ or a token which is causally dependent on $u$.

This definition of causal dependency is formulated for a path consisting of processes and queues only. If a channel $C_i$ of the path is a register, the tokens in this register are only considered to be causally dependent until they are first read by process $P_{i+1}$ in order to prevent that due to the non-destructive register read access several executions of $P_{i+1}$ produces causally dependent tokens. The reason for this exception becomes clear when looking at the modeling intentions of latency path constraints in the following.

In general, a latency path constraint limits the maximum or minimum time the system needs to respond to a certain event (e. g. arrival of input data), resulting in a constraint for a causal sequence of process executions. In the SPI model with its data rate intervals resulting in a multi-rate system with possibly uncertain relative execution rates of processes, this causal sequence of process executions is most efficiently represented by a sequence of causally dependent tokens. This representation is valid for queue communication semantics due to the fact that each token is produced and consumed exactly once and thus establishes a one-to-one correspondence between process executions. However, a token written to a register can be read several times, possibly infinitely often, resulting in a possibly infinite sequence of process executions and thus in a possibly infinite response time. Typically, the time of the first response is critical and to be constrained (e. g. a switch to a failure operation mode of a control system in response to a failure signal from the controlled system). Thus, for tokens on registers the first access is the one to be constrained. This is accounted for by the exception for register tokens in the above definition of causal dependency.

Given a schedule of the system, a constructive method can check if a latency path constraint is satisfied. This method computes for each token in the first channel of the path all possible values for the time difference between its creation and the time a causally dependent token is consumed from the last channel of the path. Based on these values, the satisfaction of the latency path constraint can be easily checked.

**Method 1** ***(Latency Path Constraint Check)***
*Assume a path* $path = (P_1 \rightarrow C_1 \rightarrow P_2 \rightarrow \ldots \rightarrow C_n \rightarrow P_{n+1})$*, a latency path constraint* $LC_{path} = [t_{path,min}, t_{path,max}]$*, and a schedule determining the timed system behavior. For all tokens* $u$ *in channel* $C_1$ *independently compute a set of time values* $\Delta_u$ *as follows:*

1. *Assume that the token* $u$ *is written to* $C_1$ *at some time* $t_u$*. Then, the token is marked with a time stamp* $t = t_u$*.*

2. *Assume that a token marked* $t_u$ *is consumed by process* $P_i$ *with* $1 < i \leq n$*. Then the tokens which are written to* $C_i$ *by the same process execution are marked with* $t_u$ *as well. If channel* $C_{i-1}$ *is a register, the time stamp of the read token is removed.*

3. *If a token marked* $t_u$ *is consumed by process* $P_{n+1}$ *from the channel* $C_n$ *at some time* $t$*, then the time difference* $t_\delta = t - t_u$ *is added to* $\Delta_u$*. If channel* $C_n$ *is a register, the time stamp of the read token is removed.*

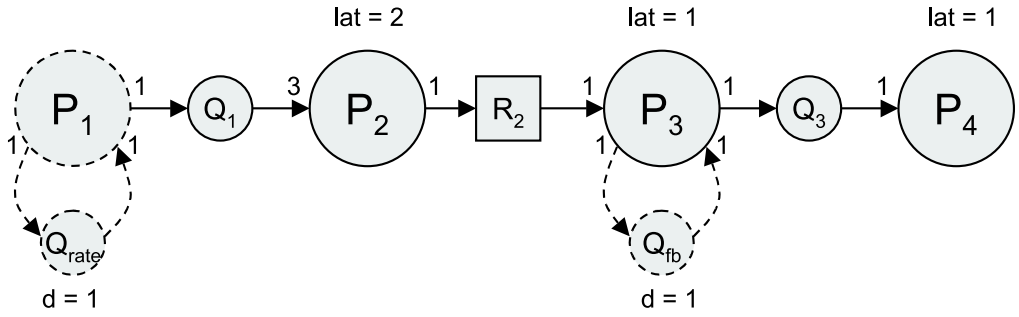



Figure 4.15: Chain of SPI processes for demonstration of latency constraints

*The constraint $LC_{path}$ is satisfied if*

$$\forall t_\delta \in \Delta_u : t_{path,min} \leq t_\delta \leq t_{path,max}$$

□

The sets $\Delta_u$ need to be computed independently, either one at a time or in parallel using a more sophisticated marking scheme, in order to prevent the interference of dependency paths of different tokens in channel $C_1$. Furthermore, it is important that the computation considers all possible system executions that may occur due to non-determinism of function, timing, or input data. This may be achieved by looking at all possible system executions (for simple cases) or by looking at corner cases guaranteeing to dominate all possible system executions. The latter is similar to assuming a simultaneous release of all processes as a worst-case scenario in rate-monotonic analysis [71].

The method states that the sets $\Delta_u$ need to be computed for *all* tokens in the first channel of the path. Evidently, then for systems periodically processing large amounts of input data that can be viewed as infinite (e. g. video filter in surveillance system), the set computation does not terminate. Thus, ways have to be found to bound the number of tokens for which the set $\Delta_u$ has to be computed. This bound depends on the used scheduling algorithm. For example, for a periodically repeating static schedule, it is sufficient to consider all tokens produced in one period of the schedule, typically called the macro period, as the scheduled system is in the same state at the beginning of each macro period. For other scheduling algorithms, the determination of this bound may be more complex.

For a demonstrational example consider the process chain depicted in Figure 4.15 and the latency path constraint

$$LC_{(P_1 \to Q_1 \to P_2 \to R_2 \to P_3)} = [0, 6]$$

denoting that time difference between the time a token is written by process $P_1$ to queue $Q_1$ and the time a token that is causally dependent on this token is read from register $R_2$ may not be larger than $6$ time units. The intention of this constraint is to specify that at most $6$ time units after new system input data has arrived the computation performed by $P_3$ is based on this data. Thus, only the first access of the data in register $R_2$ is to be constrained justifying the special treatment of register tokens with respect to causal dependency.

In the following, the above method is applied to check the satisfaction of a constraint by a simple periodic schedule of the process chain as depicted in Figure 4.16. For brevity,

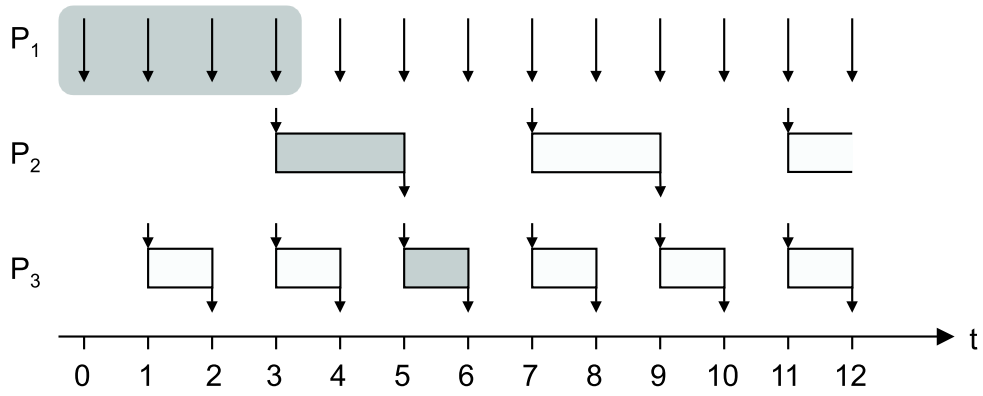



Figure 4.16: Simple periodic schedule of the process chain depicted in Figure 4.15

it is assumed that processes read at start and write at completion. Then, virtual process $P_1$ outputs a data token at all times $t = 0, 1, 2, \cdots$. Hence, at time $t = 3$, process $P_2$ reads the first three data tokens and after its latency of 2 time units writes its result to register $R_2$. At $t = 5$, $P_3$ reads the new data token from $R_2$ and starts its first execution based on the input data produced at times $t = 0, 1, 2, 3$. This execution behavior repeats cyclically with a constant offset of $4$ time units. Thus, the set of time values $\Delta$ can be computed based on those first four data tokens and results in $\Delta = \{2, 3, 4, 5\}$. As all computed time values are greater than $0$ and less than $6$ time units, this schedule satisfies the specified timing constraint.

An interesting side effect of the above constraint is that it not only imposes a constraint on the response time, but also on the maximal separation between the arrival of data tokens at queue $Q_1$. If the separation between each set of four input tokens is greater than 3 time units, the constraint can never be satisfied.

While inter-process causal dependencies are correctly accounted for by the above definition and checking method of constraints, it is assumed that each output token of a process is causally dependent on the input tokens read at the same process execution (Step 2 of constraint check method). Thus, possibly more complex intra-process causal dependencies, like an internal buffer resulting in an input-output delay in terms of one or more process executions, are not considered. However, such dependencies can be easily modeled using virtual feedback channels denoting these input-output delays.

Consider the process chain of Figure 4.15. Assume that the first computation of process $P_4$ based on new input data is to be constrained. The straight-forward way is to specify a latency path constraint

$$LC_{(P_1 \to Q_1 \to P_2 \to R_2 \to P_3 \to Q_3 \to P_4)} = [0, 7]$$

for the direct path from the input channel $Q_1$ to the process to be constrained $P_4$. Then, the constraint assumes that the first execution of $P_4$ based on the input tokens produced at times $t = 0, 1, 2, 3$ is the fourth execution starting at time $t = 6$. The corresponding set of time values computed based on the schedule depicted in Figure 4.16 is $\Delta = \{3, 4, 5, 6\}$ resulting in a positive check of the constraint.

If however, process $P_3$ has an internal input-output delay of one process execution, the first execution of $P_4$ based on the input tokens produced at times $t = 0, 1, 2, 3$ is the fifth

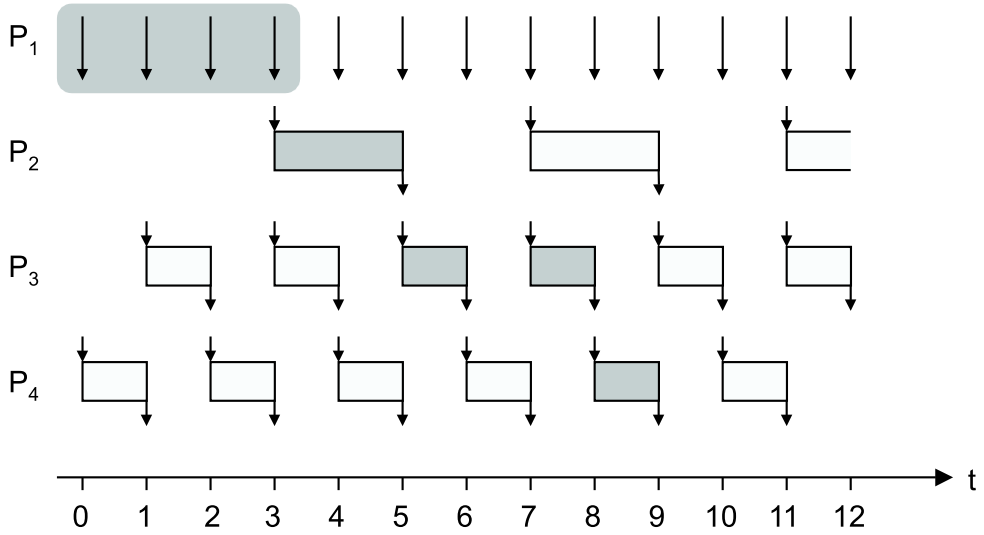



Figure 4.17: Periodic schedule of the process chain depicted in Figure 4.15 with a causality path resulting from an internal input-output delay in process $P_3$

execution starting at time $t = 8$ as depicted by the grey-shaded path in Figure 4.17 resulting in a maximum path latency of 8 time units violating the constraint. Thus, the above constraint specification results in a false acception of the schedule since the input-output delay of process $P_3$ is not accounted for. However, this can be achieved by modeling the internal delay using the virtual feedback channel $Q_{fb}$ and modifying the latency path constraint to include $Q_{fb}$ as follows

$$LC_{(P_1 \to Q_1 \to P_2 \to R_2 \to P_3 \to Q_{fb} \to P_3 \to Q_3 \to P_4)} = [0, 7]$$

Then, both process executions of $P_3$ involved in the causality chain are accounted for in the path description of the constraint and the corresponding set of time values $\Delta = \{5, 6, 7, 8\}$ results in the correct rejection of the schedule.

While it is straight-forward to use latency path constraints to limit response times as seen in the above examples, latency path constraints can also be used to model other types of timing constraints such as a rate constraint of a process or a synchronization constraint for the execution of two processes. The reason for this capability is that timing constraints in general restrict the time between two events such as data arrival or production and process start or completion. The difference to response time constraints is that in the general case there may be no causal dependency between those events and thus no data token flow that can be used to constrain the time between these events. However, using virtual channels such a token flow can be generated and latency path constraints along these channels can be used to model general timing constraints.

As seen in the example depicted in Figure 4.14, a virtual feedback channel can be used to continuously activate a process by letting each process execution produce the token needed to activate process for its next execution. By specifying a latency path constraint on this feedback channel, the time between to consecutive process executions and thus the execution rate of the process can be constrained. Figure 4.18 (a) shows a process $P$ having a virtual feedback channel $Q_{rate}$ with an associated latency constraint

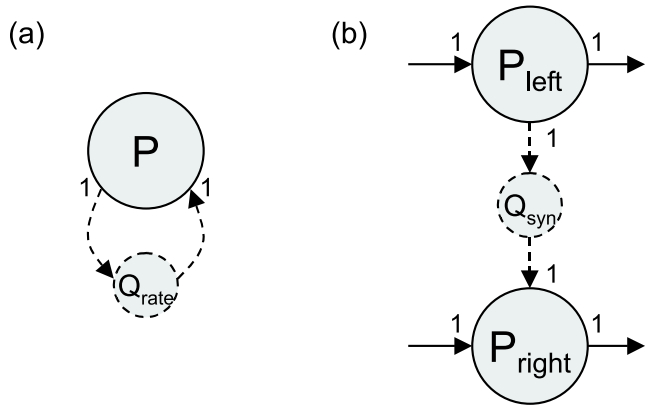



Figure 4.18: Examples for (a) rate and (b) synchronization constraints

$LC_{\{P \to Q_{rate} \to P\}} = [3, 3]$ constraining the time a token may spend on the channel (between being written to it and removed from it) to exactly 3 time units. In order to determine the rate of process $P$ however it has to be known at which time during process execution the token is written to and removed from channel $Q_{rate}$. Since $Q_{rate}$ is a virtual channel, the communication region parameter can be chosen arbitrarily. For rate constraints, it is typically set to $cr_{Q_{rate}} = 'START'$ for both accesses in order to eliminate any possible influence of the process latency on the specified execution rate. Then, process $P$ is constrained to execute with an exact period of 3 time units.

A timing synchronization between two processes that do not communicate with each other but nevertheless depend on each other can be enforced by a so called synchronization constraint (e. g. [21]). A typical example is the synchronized output of two correlated values such as the left and the right samples of a stereo audio signal. Figure 4.18 (b) shows a SPI model with processes $P_{left}$ and $P_{right}$ that produce the correlated audio samples. Both processes access the virtual channel $Q_{synch}$ at the end of their execution ($cr_{Q_{synch}} = 'COMPLETION'$). The corresponding latency constraint $LC_{\{P_{left} \to Q_{synch} \to P_{right}\}} = [0, 1]$ constrains process $P_{right}$ to complete not before but at most 1 time unit after process $P_{left}$ thus enforcing a maximum jitter between the output of both samples of 1 time unit.

There are more complex timing constraints that involve more than two events whose timing relations are constrained. A typical example is a burst constraint that specifies how many events (incoming data tokens or process executions) may occur in a certain time and what is the minimum time between the occurrences of those events. In [55], it is shown how such complex constraints can be modeled using several virtual model elements and several latency path constraints. As this representation is quite complex, it seems to be not suitable for direct specification by the designer. In contrast to other SPI constructs that can be automatically derived from the input language models of the functional specification, designer interaction is necessary for constraints as typical specification languages have no notion of constraints. Thus, [55] furthermore introduces a template-based specification of timing constraints which is more compact, more intuitive, closer to the input description, and can be easily translated to the SPI model.

#### 4.8.2.2 Power Consumption

In contrast to timing constraints, there are basically just two types of constraints concerning power consumption: either a single power consumption constraint for the whole system or a more fine-granular constraint limiting the power for the execution of a certain system function.

While the former constraint can be simply specified by a single interval, e. g., $PC_{system} = [pow_{min}, pow_{max}]$ limiting the power consumption of the whole system, the latter constraint can be defined similar to latency path constraints. Assuming that the system function to be constrained consists of several processes that are communicating with each other and is triggered by certain input data, one or more paths starting from the channel(s) containing the triggering data tokens can be defined covering all processes representing the system function. Then, the power consumption needed for the execution of the processes along these paths processing the triggering data can be constrained by another interval $PC_{function} = [pow_{min}, pow_{max}]$. An example where power constraints for certain system functions have been specified is the pico-cellular base station as depicted in Figure 4.7. Here, the different execution scenarios due to the different packet types have specific power consumption constraints [96].

#### 4.8.2.3 Other Constraints

In Section 4.3.2, the missing application dependency of properties like size, weight, and cost has been cited as the reason for not capturing them in the SPI model but rather in the augmenting architecture model. Thus, no constraints are defined for them in the SPI model as well.

Furthermore, although SPI parameters have been defined for the required memory size of processes and channels, no memory constraints are defined in the SPI model. The reason for this is that the memory size of a system is usually not directly constrained but rather is a component of other constraints concerning e. g. chip area or monetary cost. Only implementation decisions, such as the selection of a certain microcontroller with a limited amount of instruction or data memory, impose direct memory constraints. But as the SPI model is focused on capturing implementation-independent application information, no constraint for memory size is provided.

However, as already mentioned with respect to the SPI parameters, the SPI model is extensible and can be easily adapted to suit special requirements or capture additional properties and constraints.

### 4.8.3 Environment Properties

In the SPI model, system properties are captured by parameters. Environment properties however are represented by constraints on virtual elements modeling the environment. The designer has to assume the whole range of environment behavior permissible in the limits of these constraints when designing the system. However, the designer may also assume that the environment will only show permissible behavior with respect to the constraint. In this sense, constraints on model elements representing the environment are assertions for properties of the environment.

For example, consider the process $P$ and feedback channel $Q_{rate}$ of Figure 4.18 (a) but with a latency path constraint $LC_{rate} = [1, 3]$. Then the designer has the choice to execute this process with a period of 1, 2, or 3 time units or even with a period changing inside the permissible range. If process $P$ however was a virtual process, the designer would have no influence on its behavior and would have to assume all permissible behaviors of $P$ and its effects on the system.

In summary, constraints on virtual model elements are used to model complex environment behavior and environment properties while constraints on non-virtual model elements are used to specify conditions to be satisfied by a system implementation and to constrain system properties.

## 4.9 Summary and Conclusion

In this chapter, a novel abstract system model called SPI (System Property Intervals) has been presented. The SPI model is targeted at enabling system-wide analysis of non-functional properties in order to allow reliable and optimized implementation of heterogeneously specified embedded systems.

SPI is based on processes communicating via unidirectional channels having either register or FIFO queue semantics. The SPI model elements are characterized by a set of parameters instead of their functionality. The parameters capture non-functional properties of the element such as timing, power consumption, and activation. A major step towards high semantic flexibility of the model is the use of behavioral intervals for the model parameters. This allows the specification of incomplete information and thus facilitates the integration of system parts whose internal functional details are only partially known, such as legacy code. Due to the abstraction of functionality and the behavioral intervals, SPI is a non-executable model. Rather, a SPI representation of a system bounds all possible behaviors of the respective system.

While behavioral intervals allow the abstraction and clustering of different process execution behaviors, process modes support their explicit specification. Using both concepts, the degree of abstraction in a SPI representation can be controlled. The extreme cases are specifying one mode for each possible execution behavior of a process (high accuracy of modeling but exponential growth of modes with number of branches) or specifying a single behavior using uncertainty intervals (low accuracy but problem size reduction). Thus, a trade off between problem size and modeling accuracy is possible.

The concept of virtual model elements allows a seamless representation of system and system environment using the same formalisms. Furthermore, virtual model elements are used to specify additional relations between processes concerning causality and timing constraints. For example, they play prominent roles in the representation of time driven activation based on the data driven process activation of the SPI model and the flexible specification of timing constraints.

While being originally created to provide a homogeneous representation for heterogeneously specified systems, the SPI model can also be used in other contexts. Due to its variable level of abstraction, SPI is also an appealing vehicle for the analysis of single-language systems e. g. in order to validate their schedulability based on a complex real-time operating systems (e. g. ASCET and ERCOS$^{EK}$ [28]).

# Chapter 5

# The SPI Workbench

This chapter presents the SPI workbench, a research platform for the multi-language design of complex digital embedded systems. The goal of this workbench is to enable system-wide analysis of non-functional properties to allow safe and optimized implementation of multi-language embedded systems. The core of the SPI workbench is the previously introduced SPI model which serves as an internal design representation of the workbench. The key points of this chapter are

- the clarification of the intended use of the previously presented SPI model as an internal design representation,
- the presentation of the basic problems of and approaches to the transformation of specification languages to the SPI model, and
- the motivation of the value added by the SPI model and methodology to system level design of embedded systems.

After an overview of the underlying design methodology of the SPI workbench in Section 5.1, the main design steps of the SPI workbench namely the transformation of the multi-language specification into the SPI model, the system-wide analysis and optimization, and the system synthesis are discussed in the following sections.

## 5.1 Design Methodology

The discussion in Chapter 3 has identified shortcomings of existing multi-language design flows in the areas of guaranteeing non-functional system properties such as timing and the integration of system parts that do not follow a certain abstract model of computation. However, both problems are crucial in embedded system design and their importance is rapidly increasing due to the growing system complexity and a larger software share. These problems have been a major motivation for developing the SPI workbench.

The basic idea of the SPI workbench is to provide an open research platform that offers various possibilities of original contributions and supports the exchange of algorithms, demonstrators, and experiments. A key point of the SPI workbench is to provide clear, well-defined interfaces. This facilitates reuse of existing tools and solutions and thus parts of existing design flows wherever possible and allows to augment them with solutions addressing the above problems.

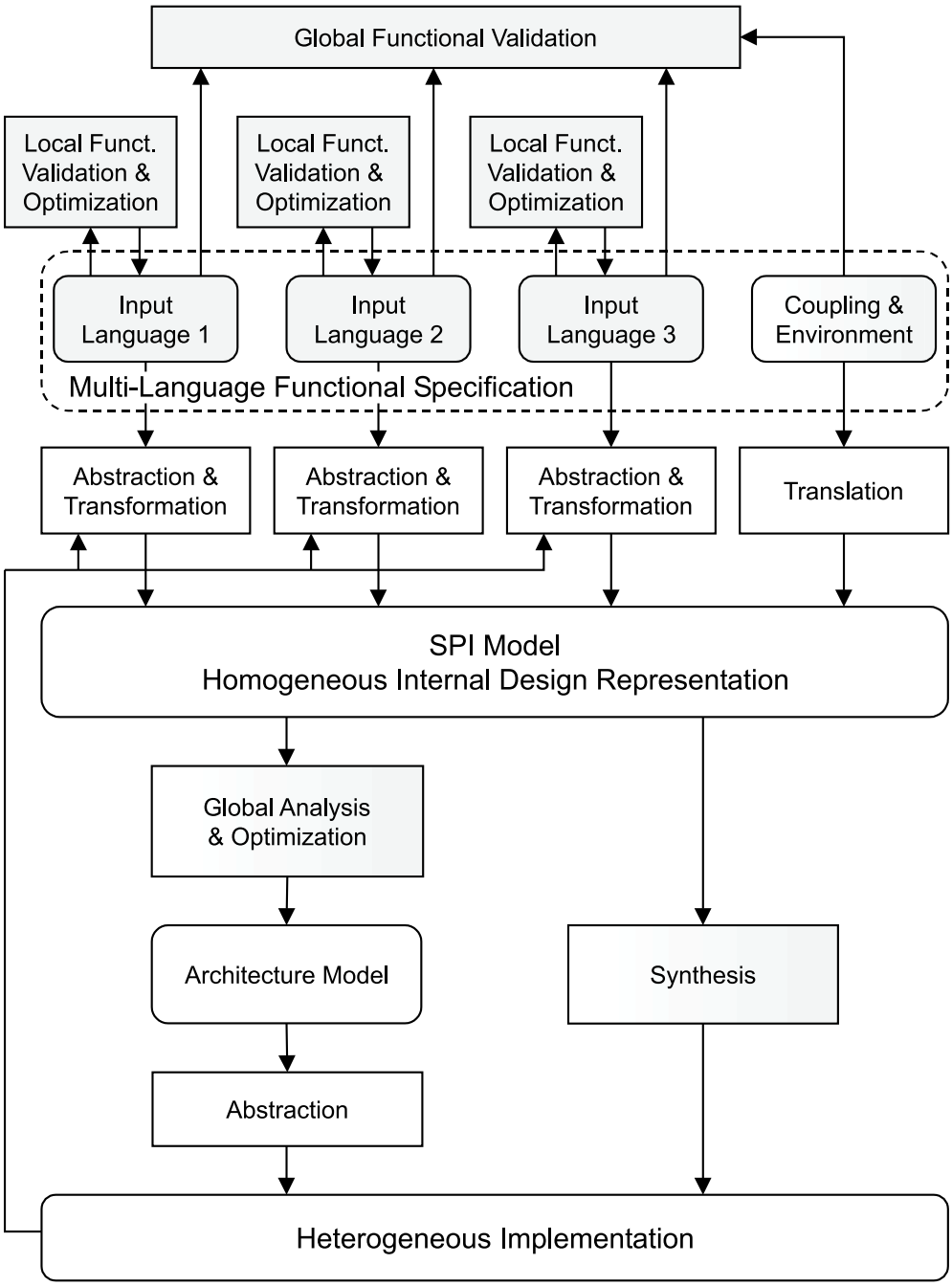



Figure 5.1: Coarse grain design flow of SPI workbench

In the following an overview of the underlying design methodology of the SPI workbench is presented. The coarse grain design flow as advocated by the SPI workbench is depicted in Figure 5.1. Design steps reusing existing tools are denoted by shaded boxes.

Input to the SPI workbench is a truly heterogeneous functional specification as known from cosimulational approaches. The different elements of this specification are:

- **Functional Blocks** The functional block descriptions are the major source of heterogeneity for the specification. As complex embedded systems integrate functions

from different domains, the system functionality is typically described using several different abstract specification languages. Here, the system functionality is partitioned into functional blocks according to the used language, i. e. each block is homogeneously described in a single language. These languages are selected because of their particular suitability for certain functions and optimizations, or because they have become generally accepted as a standard within an application field. Adding to this heterogeneity, is the fact that reused components, possibly described in yet another language or incompletely documented, like ”legacy code” or IP (intellectual property) components have to be incorporated. Furthermore, also system parts in very early design stages (incomplete specification) may have to be considered.

- **Coupling Information** Besides the different functional blocks, the specification also has to include the information on how the different functional blocks are connected. This information is typically captured by the interface descriptions for cosimulation that contain the interconnection structure (e. g., which port of block $A$ corresponds to which signal of block $B$) as well as type conversions. An example for such an interface description is the MUSIC coordination file [19] based on the SOLAR format [53].

- **System Environment and Constraints** For the design of embedded systems, the system environment and its imposed constraints are of central importance. In simulation-based specification languages (e. g., Simulink), the system environment is typically modeled together with the system in order to allow a simulation of the system in its context (e. g., a model of motor control logic together with differential equations modeling the motor). However, possibilities to specify non-functional constraints (e. g., timing constraints) are typically not provided. Thus, a suitable notation external to the specification languages has to be used. Requirements for this notation are:

  – to be semantically as close as possible to specification languages in order to allow the designer to specify the constraints intuitively,

  – to be specification language independent in order to allow the specification of constraints across language boundaries, and

  – to allow a simple translation of the specified constraints into the SPI model.

  Here, standardization efforts such as Rosetta [48] are closely monitored but currently a template-based approach [55] is used.

The advantage of such a truly multi-language functional specification as compared to an approach based on a single new language supporting several abstract models of computation is the possibility to reuse existing design environments. These design environments usually include design libraries facilitating reuse of previously designed system parts and tools for analysis and domain-specific optimization. Furthermore, designers are well-trained and experienced in how to model and specify using ’their’ design environment leading to a typically higher design quality. In the context of the SPI workbench methodology, these design environments are used not only for functional specification of system

parts but also for their local functional validation and optimization. The system-wide or global functional validation is performed using existing cosimulational approaches.

The first design step not relying on existing tools is the creation of a homogeneous internal representation of the functional specification, the SPI model. This is achieved by a separate transformation of each functional block into corresponding SPI model elements. The arrow from the implementation to this transformation step denotes the dependency of some SPI model parameters on the implementation (see Section 4.3.2). Furthermore, the information regarding the coupling of the different functional blocks and the constraints imposed by the system environment are translated to the SPI model as well. Here, the coherence of the cosimulation coupling and the coupling on the SPI level has to be ensured. Otherwise, the functional validation results based on cosimulation are not valid for the implementation based on the SPI model. This coherence can be achieved by a well-defined automatic or at least systematic derivation of cosimulation interfaces and SPI model elements for the coupling of functional blocks from the same coordination file. This can be based on the experience and methods for the automatic generation of consistent interfaces for cosimulation runs at different levels of detail as well as for implementation [94].

Analogous to the SPI model being the abstract representation of the functional specification or application, there is an architecture model that represents the target architecture. Both models have to match in terms of granularity and abstraction, i. e. as the application is modeled in terms of processes and channels the target architecture has to be modeled in terms of processing elements, memories and buses and not, e. g., on the register-transfer-level. Additionally, the architecture model also captures design decisions such as mapping of SPI elements to resources and chosen scheduling parameters such as process priorities. A detailed description of the architecture model can be found in [64].

Based on these two models, system-wide or global analysis and optimization is performed. Results of this step such as a clustering of SPI processes or the mapping of a SPI channel to a certain bus of the target architecture are back annotated to the respective model denoting an inner design iteration. Furthermore, an outer design iteration characterized by feedback from global analysis and optimization to local optimization and model transformation exists but is not depicted in Figure 5.1 for reasons of clarity. Global analysis and optimization involves both newly developed as well as existing tools, the exact characteristics of this macro design step, however, are heavily dependent on the properties of application and architecture.

Similar to the global analysis and optimization, the synthesis step is based on the SPI model as well as on the architecture model and involves existing as well as new tools. The synthesis step in the SPI workbench can be divided into two interdependent tasks, the host synthesis mainly based on existing tools (e. g., code generators) generating functional blocks and the coordination synthesis relying only partly on existing tools creating interfaces and scheduler.

The following sections describe the novel design steps of the SPI workbench methodology in more detail.

## 5.2 Input Transformation

In Chapter 3, a homogeneous internal system representation has been identified as a main prerequisite for performing efficient system-wide analysis and optimization. In case of the SPI workbench, this representation is the SPI model that has been presented in Chapter 4. This section describes how the SPI model is created from the heterogeneous multi-language specification that is the input of the SPI workbench. The generation of the SPI model can be divided into two steps, the transformation of the functional blocks and the translation of the coupling information and constraints.

For each functional block of the specification, a SPI representation is created separately. The key of this transformation is to represent the model of computation of the functional block using SPI model elements. This can be easily achieved for a functional block specified in a language with an underlying model of computation which is closely related to the SPI model of computation, e. g., data flow process networks. However, for models of computation which are based on assumptions that are targeted to simulation (e. g., computation or communication in zero time) this requires a slight adaptation of the model of computation in order to provide a SPI representation that is meaningful in the context of synthesis. This adaptation may be called a standard implementation interpretation of the respective model of computation and clearly may not violate the model causality which would result in the production of different outputs. An example for such a model adaptation is the transformation of MATLAB/Simulink to SPI as described in Section 6.2.2. The Simulink transformation even goes further and relaxes implementable internal timing requirements in order to increase the design space.

In general, the functional block transformation involves the following steps:

- The elements of the functional block description, in the following called input language, are mapped to corresponding SPI elements. This is typically a template-based approach where a parameterized template consisting of SPI elements is provided for each input language element or group of elements. This mapping is subject to designer input concerning the granularity of the generated SPI elements. This is necessary in order to obtain a process granularity suitable for the SPI model that assumes processes to represent complex functions such as an FFT rather than operations such as an addition. In case of the Simulink transformation, the designer can provide a file specifying which Simulink blocks have to be clustered and mapped to a single SPI process [57]. For subsequent steps, an input reference is stored for each SPI model element specifying which input language elements correspond to the SPI element.

- The parameters of the created SPI elements are extracted from the corresponding input language elements. The token size parameter of a certain SPI channel for example is determined by analyzing the data types that are communicated over this channel. For process parameters, the level of abstraction used in the parameter extraction can be controlled using the concept of process modes. The extreme cases are specifying one mode for each possible execution path of a process (high accuracy of modeling but exponential growth of paths with number of branches) and specifying a single behavior using uncertainty intervals (low accuracy but problem size reduction). Thus, during parameter extraction, a trade off between problem size

and accuracy of modeling which is directly related to the accuracy of the results is possible.

As the determination of implementation-dependent SPI parameters is only possible and meaningful when the target architecture is known, it is typically performed later in the design flow during system-wide analysis and optimization.

- As specification languages often allow the modeling of both system and system environment, the designer has to provide information on whether a certain input language element models the system to be implemented or the system environment. Based on this information, the virtuality flag of the corresponding SPI element has to be chosen accordingly.

Furthermore, the other elements of the system specification have to be transformed into the SPI model. The translation of the information regarding the functional block coupling, typically provided in a cosimulation coordination file (e. g., [19]), yields SPI elements glueing the SPI representations of the functional blocks together. As already mentioned above, the consistency of this glue code with the cosimulation can be achieved by systematically generating it from the same coordination file. This is the same way already used in cosimulational approaches in order to generate consistent interfaces for simulation on different levels of abstraction as well as for implementation.

Regarding constraints, currently only the translation of timing constraints is supported. Since the specification of timing constraints is currently based on a proprietary file format which was designed to allow an easy translation to the SPI model the template-based translation is straight-forward [55].

As the coordination file as well as the constraint specification refer to identifiers (e. g., block or signal names) of the input languages, the translation of this information relies on the stored input reference in order to determine the corresponding SPI model elements.

The result of the input transformation step is a homogeneous design representation of the complete system. This representation can be seen as a process network with the SPI model serving as coordination language and arbitrary host languages where SPI captures the interaction of processes inside of functional blocks as well as the coupling of the different functional blocks. The host language of a SPI process is given by the language of the corresponding input language elements as defined by the input reference.

Due to the focus of the SPI model, the choice of the host languages is not limited by the need to being executed or simulated together. The only restriction on the host languages from a modeling point of view is that the extraction of SPI parameters has to be possible. This facilitates the incorporation of IP blocks into the design flow as IP providers do only need to provide a SPI characterization of their IP block and do not need to disclose functional details.

The dualism between the SPI model (dark gray shaded elements) and host languages (light gray shaded elements) is shown in Figure 5.2 which depicts the flow of the input transformation step. However, due to a possible granularity adaptation, the host language in context of the SPI model may also include coordination language elements of the input language. An example for this are the upper two elements of functional block 2 which are clustered and represented by a single SPI process.

In this work, examples for input transformations are shown for Simulink (Section 6.2.2), software processes with generic communication functions (Section 6.2.1), processes ad-

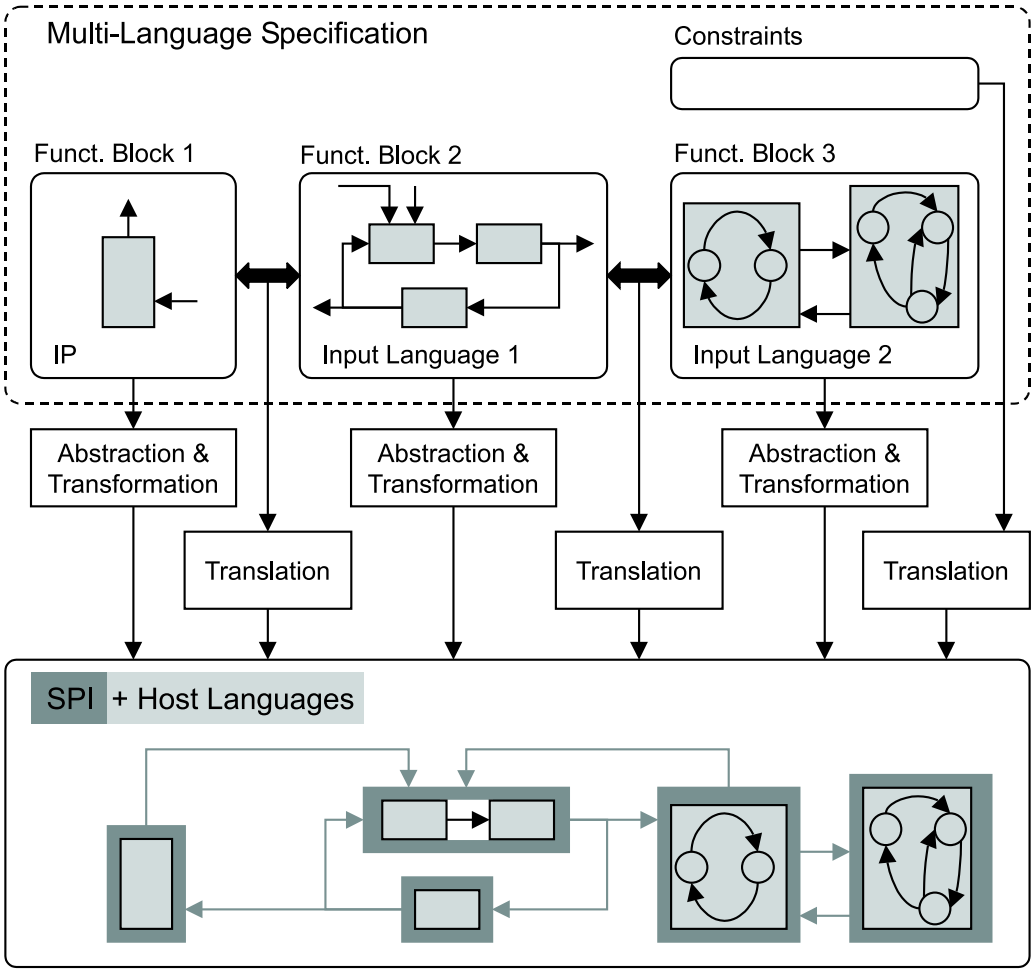



Figure 5.2: Transformation of the heterogeneous specification to the SPI model

hering to a typical RTOS execution scheme (Section 6.1.2), and state-based models (Section 6.1.4). Furthermore, the principle of a transformation has been theoretically shown for Kahn process networks [92], hardware description languages and StateCharts [80], and SDL [44].

## 5.3 System Analysis and Optimization

The main added value of the SPI workbench methodology in comparison with other approaches is the possibility to perform system-wide analysis and optimization with respect to non-functional system properties. During this step, the implementation of a system is defined and optimized and the satisfaction of constraints is validated. For this task the SPI model capturing the application or functional system specification has to be augmented with a model of the target architecture.

This architecture model is not part of this work. However, its required characteristics follow directly from the extent of the SPI model and thus are briefly summarized in the following. The architecture model has to match the SPI model in terms of granularity and

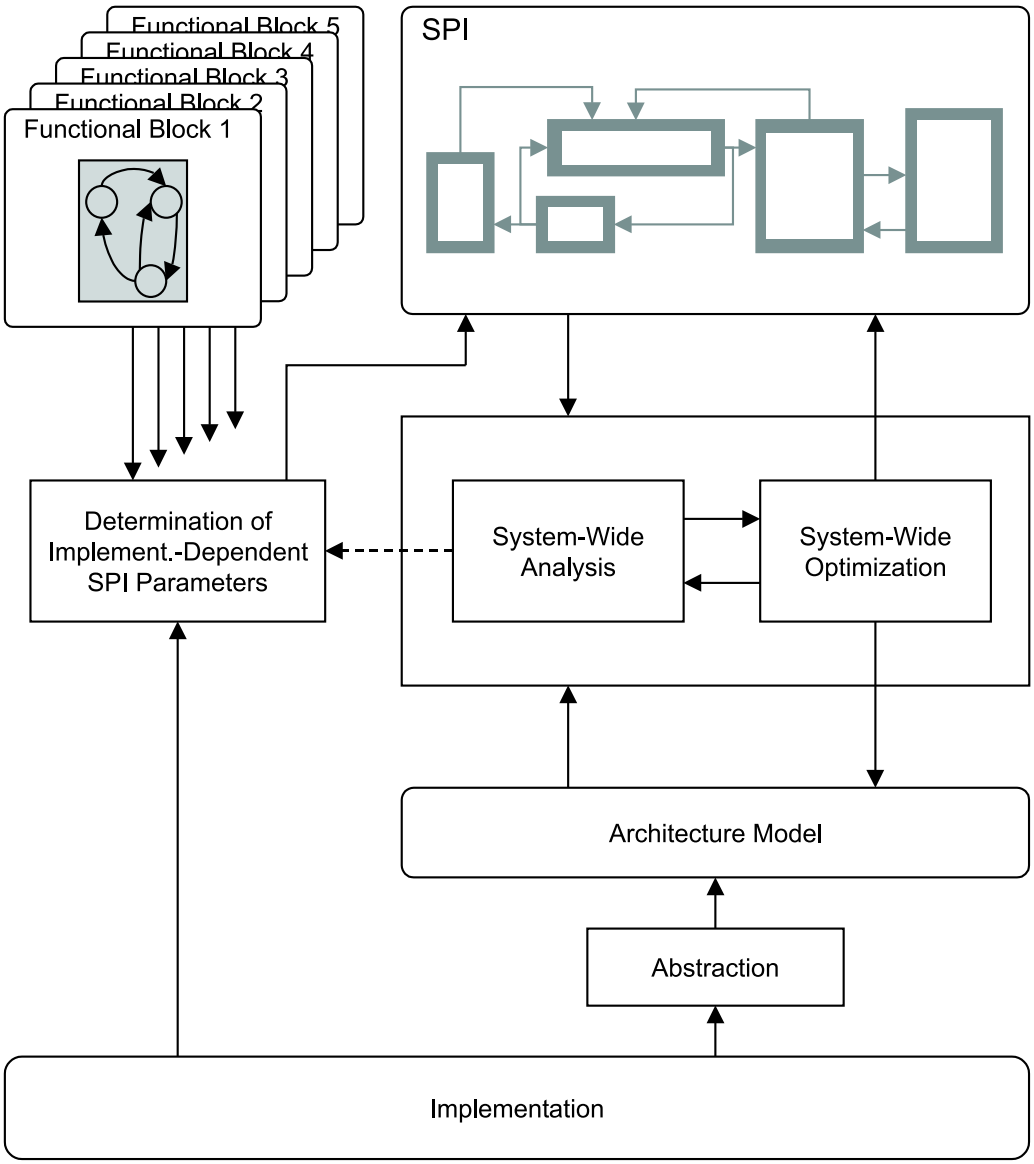



Figure 5.3: System-wide analysis and optimization with determination of implementation-dependent parameters

abstraction. The matching granularity is achieved by capturing the target architecture at the component level i. e. in terms of processing elements, memories, and buses. The different elements are to be described at the same level of abstraction as the SPI model by parameters representing architectural properties such as cost, power consumption, memory, bandwidth, weight, and size. Furthermore, the software architecture including real-time operating system, drivers, task switching times, etc. as well as scheduling and arbitration issues have to be accounted for. The architecture model used in the SPI workbench is described in more detail in [64].

The architecture model, however, is not the only architecture representation in the SPI workbench design flow. The determination of implementation-dependent SPI parameters is based on an instruction-level or cycle-true representation (e. g., a simulator) of the ar-

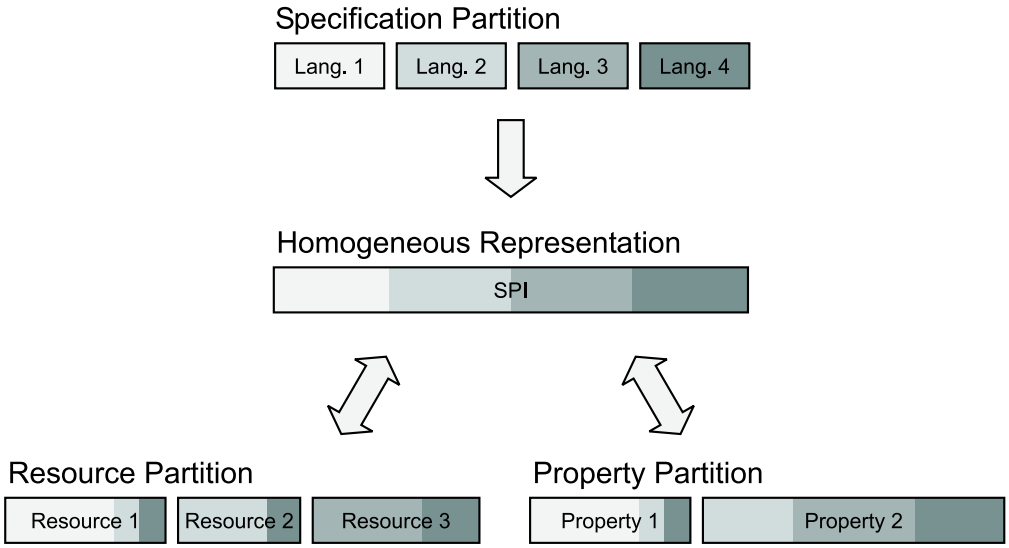



Figure 5.4: System partitions according to different system design aspects

chitecture (see Section 6.2.1). While intuitively a part of the input transformation step, the determination of implementation-dependent parameters is only meaningful with an implementation in mind. Thus although being local to the SPI element, the determination is performed during the system-wide analysis and optimization step where the implementation is defined. An additional reason is the desired accuracy of the parameters depending on the system-wide analysis or optimization step. For example, rough estimates are sufficient for a first design space exploration determining potential mappings of an application to a target architecture whereas a method validating constraints or calculating quality of service values for a final implementation requires conservative and precise bounds on a parameter value. The flow of information regarding implementation-dependent parameter extraction is depicted on the left side of Figure 5.3 where the dashed arrow denotes the control of the extraction accuracy.

For complex embedded systems, the SPI model and the architecture model together span a huge design space, for which there is not a single best analysis or optimization method. This is similar to the fact that there is not a single best specification language for embedded systems. At first, this seems like a contradiction to the concept of a homogeneous representation for the whole system. However, when examining existing analysis and optimization methods or styles, the benefit of a homogeneous design representation can be easily motivated.

Analysis and optimization methods make assumptions on certain properties in order to obtain their results. Thus, they are only applicable to a system or system part if it satisfies these assumptions. Examples are the assumption of independent periodic processes made by rate monotonic analysis [71] or constant data rates for methods targeted to synchronous dataflow process networks. Thus, in order to apply these methods to a system or system part, the language the system is specified in is not necessarily of importance. For example, most systems specified in Simulink can be scheduled using an existing SDF scheduling method after being transformed to the SPI model [57]. In this context, the

benefit of a homogeneous design representation is that regions or system parts, having the desired properties for the application of a certain design method, can be identified regardless of language boundaries. Then, the system can be partitioned into regions of similar properties instead of being partitioned according to the used languages. An example for a such property partitioning method is the `SDFIslandFind` method [13] which identifies regions of SPI representations having synchronous dataflow (SDF) semantics. The rationale behind this property partitioning is the reduction of dynamic scheduling overhead by clustering the identified regions to larger SPI processes based on an SDF scheduling method.

Similarly, after a mapping of functional elements to architectural resources has been defined, the system can be partitioned according to resource boundaries. This partitioning reflects a certain analysis style where the system is analyzed resource by resource and the analysis results of one resource form constraints for the other resources [82]. Then, the homogeneous representation provides a formal underpinning for the combination of different analysis methods and their interfacing at partition boundaries [84].

Figure 5.4 shows an abstract example which summarizes the above ideas and motivates the benefit of a homogeneous implementation for system-wide optimization and analysis. The example deals with a system specified using four different languages denoted by different shades of gray. The figure shows three partitions of the system according to different design aspects. The first partition is the specification partition where the system is partitioned according to the used languages. These languages are selected based on the convenience of a certain language to specify a certain function. The property partition shows that the system is partitioned according to two property sets important to the application of certain analysis and optimization methods, whereas the system is mapped to three different resources denoted by the resource partition. As can be seen by the distribution of the different shades of gray, all partitions are different from each other with respect to the distribution of the system parts. The homogeneous representation unifies all system parts and provides a means to transform the system between different partitions. This way, the application of design methods is no longer limited by language boundaries but can be based on system partitions defined by similar model properties or resource boundaries.

The flow of information in the system analysis and optimization step is depicted in Figure 5.3. It can be seen that the global analysis and optimization methods are based on the SPI model (without host languages) and the architecture model alone and do not require additional information regarding application or target architecture. The feedback arrows from the optimization box denote back annotations such as, e. g., taken implementation decisions to the architecture model or the result of a clustering method leading to a granularity change (merging of several elements) in the SPI model. Although the depicted flow clearly assumes a given target architecture (platform-based design), the selection of a suitable target architecture can be incorporated in the design flow and can be denoted graphically by a feedback arrow from the optimization to the implementation.

Examples of analysis and optimization methods including a variety of scheduling and system-level timing analysis approaches are described in Section 6.3.

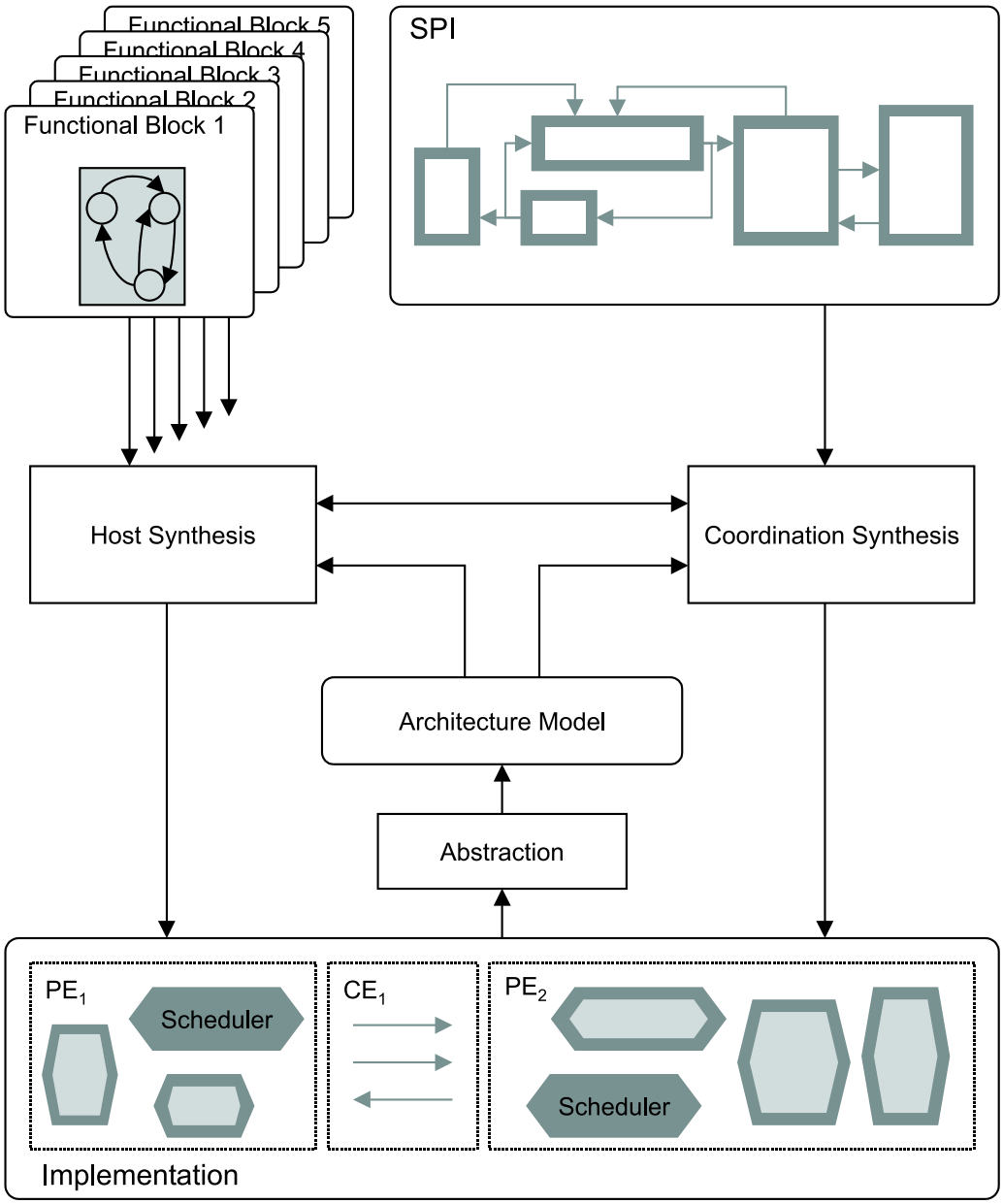



Figure 5.5: Synthesis step divided in host and coordination synthesis

## 5.4 Synthesis

The generation of a system implementation based on the design decisions taken in the preceding system-wide optimization step is the task of the synthesis step. Unlike the usual split in hardware and software synthesis, this step can be divided into two tasks, the host synthesis and the coordination synthesis. The design flow of the synthesis step is depicted in Figure 5.5.

The host synthesis generates for each SPI process separately a functional block based on its corresponding host language description. Here, an existing synthesis path from the host language to an implementation (processor or hardware) including standard code

generators and synthesis tools is reused with as little adaptation as possible. The existence of such a synthesis path is a prerequisite for the possibility to map a SPI process to a certain resource. This constrains the design space and is an example of an implementation constraint that has to be represented by the architecture model in order to prevent that unsynthesizable mappings are defined during system optimization.

The applicability of this approach and the ability to reuse code generators in particular has been shown for the Simulink to SPI translation where the Real Time Workshop code generator of TheMathworks [73] has been adapted to produce a C function for each SPI process according to a granularity file specifying which Simulink blocks are represented by each SPI process.

The coordination synthesis generates the glue code for the functional blocks, generated by the host synthesis, in order to execute together according to the SPI execution model. Different from the usual naming conventions, this step is intentionally called coordination instead of interface synthesis as not only the communication but also the scheduling is implemented. Thus on one hand, communication primitives and lean process wrappers providing the functional blocks with access methods for the communication primitives are generated based on the SPI communication elements. And on the other hand, coordination constructs such as a scheduler or an RTOS configuration file are generated based on the implementation decisions represented in the architecture model.

A possible way to reuse existing work in this area (e. g., cosynthesis or interface synthesis tools such as ArchiMate [1] or CoWare N2C [20]) is to provide an output to a system-level implementation language such as SpecC or SystemC. Provided that the host language of all SPI processes is C or can be translated to C by a code generator, this can be achieved by generating communication primitives and wrappers in the particular system-level implementation language. Then, the existing tools can be scripted in order to generate the implementation according to the design decisions taken in the optimization step.

Both tasks, host and coordination synthesis, are interdependent as their outputs have to work together and thus, have to communicate identifiers and changes due to internal optimizations between each other. However, their differentiation reflects a separation of concerns analogous to the differentiation between host and coordination language.

## 5.5 Summary and Conclusion

In this chapter, the SPI workbench, a research platform and methodology for the multi-language design of complex embedded systems, has been presented. Starting from a truly multi-language specification containing abstract specification languages as well as implementation languages, the system is transformed into an abstract internal design representation, the SPI model, which serves as basis for system-wide analysis and optimization.

A fundamental principle of the SPI workbench is to integrate existing design environments and tools wherever possible. Thus, instead of being a novel design environment, the SPI workbench rather can be seen as a platform augmenting existing cosimulational design flows with formal analysis capabilities regarding non-functional system properties such as timing or power consumption.

In this context, a crucial point for the success of the SPI workbench is the availability

of well-defined interfaces to existing design flows and the availability of input language transformations in particular.

The SPI workbench is a compositional approach as it is based on a single homogeneous internal design representation, the SPI model. By assuming a truly heterogeneous specification while nevertheless providing a close formal integration supporting system-wide analysis and optimization, the SPI workbench combines the advantages of both cosimulational and compositional approaches.

# Chapter 6

# Application Examples

After the presentation of the SPI model and its application to compositional design in the SPI workbench, this chapter provides a set of examples demonstrating the application of the SPI model. The first two sections provide modeling examples showing how different design concepts can be represented by the SPI model as well as examples on how properties can be extracted from input languages. Afterwards, the application of scheduling and analysis methods are presented in order to show the usefulness of the SPI approach.

## 6.1 Modeling

This section provides some modeling examples in order to show how the SPI model can be used to represent constructs or models of computations and their properties which are not intuitively covered by the constructs of the SPI model.

### 6.1.1 Relative Execution Rates

Models of computation differ in the permissible relative execution rates of processes. Different possibilities are single-rate execution where all processes execute at the same rate (e. g. marked graphs [18]), and multi-rate execution where the relative execution rate of processes is given by a ratio of sample times (e. g. Simulink [74]) or data rates (e. g. synchronous dataflow [68]).

In the SPI model, the relative execution rate of a pair of processes in general is undefined. For example, consider processes $P_1$ and $P_2$ of Figure 6.1 (a). If channel $C$ is a register, no information on the relative execution rate is given as register communication is unsynchronized. However, if processes $P_1$ and $P_2$ are communicating over a queue their average relative execution rate is given by their respective data rates $s_C$ and $r_C$ on this queue and evaluates to $s_C/r_C$ meaning that $P_2$ is executed $s_C$ times for $r_C$ executions of $P_1$. The resulting relative execution rate is an average rate as the buffering of data tokens decouples the executions of both processes.

For constant data rates, this results in a constant relative execution rate (equivalent to synchronous dataflow). For data rate intervals however, several different average relative execution rates are possible. For data rates of e. g. $s_C = [1, 4]$ and $r_C = [2, 3]$, the different possible relative execution rates of the example process pair are bounded by $s_{C,min}/r_{C,max} = 1/3$ and $s_{C,max}/r_{C,min} = 2$.

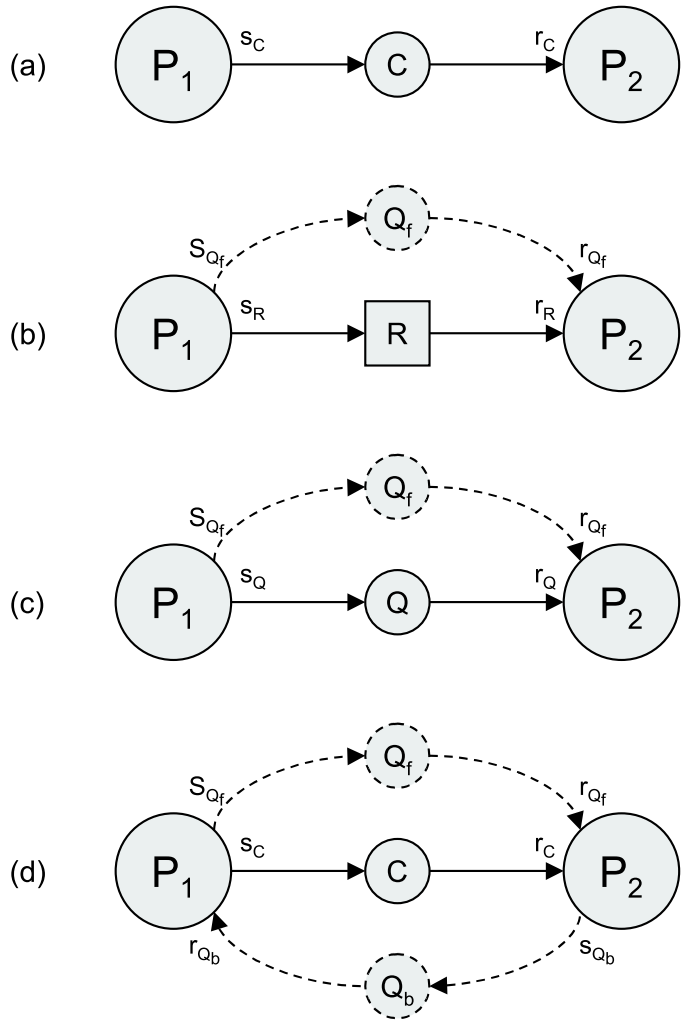



Figure 6.1: Relative execution rates: Example process pair (a) with a synchronizing virtual queue (b) and a back queue constraining the maximum iteration offset (c)

In general, a relative execution rate of two processes given by the input language can be represented in the SPI model by a virtual queue with constant data rates synchronizing the sending and receiving process regardless of their real communication i. e. even if they do not communicate. This is shown in Figure 6.1 (b) for two processes communicating over a register. For example, a relative execution rate of $1$ could be enforced by choosing $s_f = r_f = 1$. Clearly, the activation function of the receiving process has to be adapted to consider the tokens on the virtual queue.

Such a virtual queue may also be used for processes communicating via a FIFO queue (see Figure 6.1 (c)). Then, it may specify a relative execution rate which is inherent in the application but is lost due to the data rate interval abstraction. However, as the ratio of input and output data rates on the introduced virtual queue ($s_f$, $r_f$) and on the queue for real communication ($s_Q$, $r_Q$) differ (otherwise the virtual queue would not have been introduced), the ratios may contradict leading to an incorrect model. For example, if constraining the relative execution rate of processes $P1$ and $P_2$ to be 3 by $s_f = 3$

and $r_f = 1$, this would contradict real data rates of $s_Q = [1,4]$ and $r_Q = [2,3]$ as three executions of $P_2$ consume at least 6 data tokens from $Q$ while one execution of $P_1$ produces at most 4 tokens on $Q$. Unfortunately, due to the non-determinism inherent in data rate intervals there is no possibility to reliably analyze whether the specified relative execution rate complies with the actual data consumption and production behaviors of the processes. The only possible analysis is the containment analysis checking whether the specified rate is contained in the set of rates possible due to the data rate intervals (as performed above). This is an example where SPI has to rely on the assumption of correct models as mentioned in Section 4.1.

Still, the first process may be executed infinitely often before the first execution of the second process. In some models of computation, relative execution rates are specified in order to ensure a synchronization of communication with register semantics (e. g. Simulink). In these cases, the reading process has to complete execution before the writing process overwrites the data the process is supposed to read. This can be modeled by an additional virtual back channel $Q_b$ as depicted in Figure 6.1 (d) with $s_b = r_f$ and $r_b = s_f$ and a suitable amount of preassigned data tokens on both virtual queues ($(d_{Q_b} + d_{Q_f} \geq \max(r_f, r_b)) \wedge ((d_{Q_b} \geq r_b) \vee (d_{Q_f} \geq r_f))$). These restrictions are necessary to ensure bounded memory and liveness [68]. Then, by choosing e. g. the parameters $s_b = r_f = 2$, $r_b = s_f = 1$, and $d_{Q_b} = d_{Q_f} = 1$, the execution sequence $P_1P_2P_1P_1P_2P_1P_1P_2\ldots$ and thus a relative execution rate of $1/2$ is enforced.

In summary, virtual queues can be used to specify relative execution rates of SPI processes that are either lost by the data rate interval abstraction or not represented as the corresponding processes do not communicate via a queue.

### 6.1.2 Time-Driven Activation

The SPI activation function presented in Section 4.5 is based on data availability, i. e. a process is activated if on all its input queues there are sufficient data tokens for an execution. Obviously, this activation principle can represent other data-driven and event-driven activation principles. In this section, it is shown how different time-driven activation principles are represented in the SPI model.

In Section 4.8.2, the specification of rate constraints by a latency constraint over a virtual feedback channel is demonstrated. Then, the process writing and reading from this feedback channel is constrained to execute periodically. By basing the activation of this process on the feedback channel, the process is periodically activated and thus time-driven. An example is process $P_p$ in Figure 6.2 which accesses queue $Q_p$ at process start ($cr_{Q_p} = 'START'$) and is activated by the presence of one token on $Q_p$. Then due to the latency constraint $LC_{\{P_p \to Q_p \to P_p\}} = p$ on queue $Q_p$, $P_p$ is activated and executed with an exact period of $p$ time units.

Exact periodic activation however is often unnecessarily restrictive if a process may execute with a bounded jitter i. e. may start execution within a certain time after being activated. This is called *relaxed periodic activation* and shown for process $P_{rp}$ in Figure 6.2 which reads one token per execution at process start ($cr_{Q_j} = 'START'$). With process $P_p$ being executed with a period of $p$ and producing a token on $Q_j$ at process start ($cr_{Q_j} = 'START'$), process $P_{rp}$ is executed with a period $p$ and a maximum jitter of $j$. The jitter is bounded by the latency constraint $LC_{\{P_p \to Q_j \to P_{rp}\}} = [0, j]$ on queue $Q_j$.

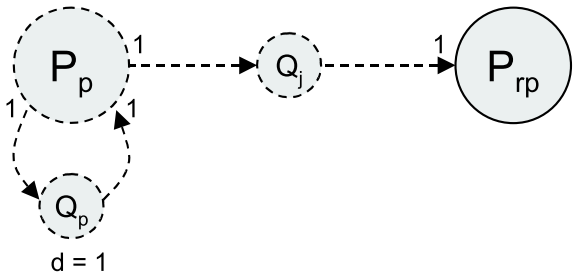



Figure 6.2: Relaxed periodic activation: Process $P_{rp}$ is executed with a period $p$ and a maximum jitter of $j$.

By changing the time process $P_{rp}$ reads the token from queue $Q_j$ to $cr_{Q_j} = 'COMPLETION'$, the meaning of the jitter constraint is changed to a deadline constraint. This is due to the fact that the constraint no longer relates to the start but to the completion of process $P_{rp}$. Then, process $P_{rp}$ is activated with a period of $p$ time units and has a deadline of $j$ time units relative to activation, i. e. $P_{rp}$ has to complete within $j$ time units after activation. This is a typical activation principle used for real-time applications where processes are activated periodically but have a constrained mobility interval (from activation to deadline) to complete their execution. For example, rate monotonic ($j = p$) [71] and deadline monotonic ($j < p$) analysis [86] assume this activation principle. The ability to model this activation principle allows the representation of many RTOS-based applications using the SPI model.

### 6.1.3 Communication Concepts

In Section 4.6.3, the basic communication constructs of the SPI model have been discussed. This section shows how to represent communication with not directly supported properties namely bidirectional and synchronous communication or more complex communication mechanisms like rendezvous, client-server, or polling communication by a combination of these basic communication constructs.

While all communication in SPI is based on unidirectional channels, bidirectional communication can be modeled easily by two SPI channels in opposite directions.

Synchronous communication defines that read and write operations must occur at the same time. In the SPI model, all communication is buffered. However, the destructive read access on queues ensures a synchronization between sending and receiving processes in the sense that data can not be read before it has been written and before all data written earlier to the queue has been read. While a strictly synchronous communication can not be represented due to the buffers, at least a synchronization in terms of forcing the writing process not to write before the reading process is ready and not to continue execution before the communicated data is read. This is called *rendezvous* communication. As SPI processes are not allowed to wait for some event (e. g. the arrival of data tokens) during execution, the representation of rendezvous communication involves the splitting of the communicating processes. Assuming e. g. process $P_w$ wants to communicate data to process $P_r$ by a rendezvous mechanism, the corresponding SPI representation is depicted in Figure 6.3 where both processes are already splitted such that the read operation is the first

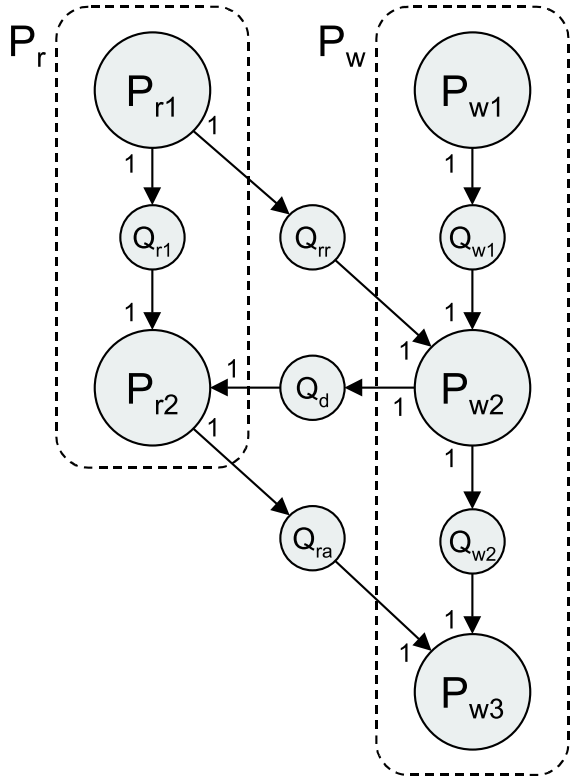



Figure 6.3: SPI representation of rendezvous communication

operation of $P_{r2}$ and $P_{w2}$ essentially consists of the write operation. By sending a token on queue $Q_{rr}$, process $P_{r1}$ signals its entrance of the rendezvous region. Process $P_{w2}$ and thus the write operation is activated by this token and writes the data to be communicated to queue $Q_d$. This in turn activates $P_{r2}$ which reads the data and signals its exit of the rendezvous region by sending a token on queue $Q_{ra}$ activating process $P_{w3}$ and thus process $P_w$ to resume execution. Other handshake-based communication or semaphore-protected variable access can be modeled similarly.

The popular *client-server* communication mechanism consists of a process (the client) requesting a service from another process (the server). This behavior is contained in the above rendezvous example as process $P_r$ (by its part $P_{r1}$) requests a service (the sending of data) from process $P_{w2}$. Again, the client process has to be split (into two processes directly after the service request) in the SPI representation as SPI processes may not wait for an external event (the response to the service request).

Two widely used communication mechanisms to react on (external) events are *interrupt* or *polling*. While the SPI process in its general form is equivalent to an interrupt mechanism (token arrives $\Rightarrow$ process is activated), it is also possible to explicitly model polling, i. e. time-driven, active waiting for a token. Then, instead of basing the process activation ($(Q_e.num \geq 1) \mapsto \{m\}$) on the token arrival on queue $Q_e$ characterizing the event (Figure 6.4(a)), the process is periodically activated and tries to read the token on $Q_e$. This is depicted in Figure 6.4(b) and leads to a representation of process $P_p$ involving

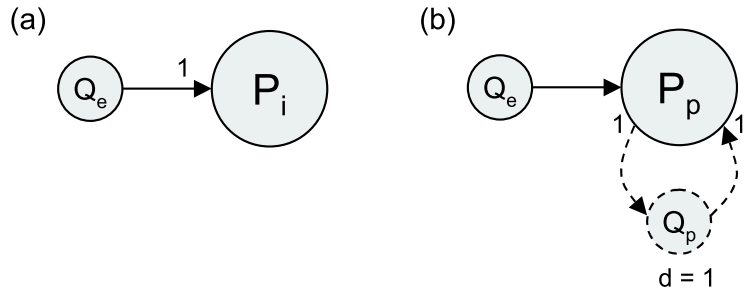



Figure 6.4: SPI representations of interrupt (a) and polling (b) mechanisms

the following modes, activation rules, and latency constraint

$$m_i = (r_{Q_p}, s_{Q_p}, r_{Q_e}, \ldots)$$

$$m_1 = (1, 1, 0, \ldots)$$

$$m_2 = (1, 1, 1, \ldots)$$

$$(Q_p.num \geq 1) \wedge (Q_e.num = 0) \mapsto \{m_1\}$$

$$(Q_p.num \geq 1) \wedge (Q_e.num \geq 1) \mapsto \{m_2\}$$

$$LC_{\{P_p \to Q_p \to P_p\}} = [t_p, t_p]$$

where $m_1$ and $m_2$ denote the failed and the successful poll, respectively. Polling can only be represented by allowing processes to access queues by a `try_receive()`-like function.

Further communication mechanisms such as an SDL FIFO queue [49] allowing the reading process random access and reordering capabilities or bounded FIFO queues in general can be modeled using a SPI process as a data object with complex access functions. An example is the general representation of an SDL queue described in [44].

### 6.1.4 State-Based Modeling

While it has been mentioned that SPI processes may have and manipulate state, the representation of this state has not been treated so far. This section shows how processes with a state-based behavior can be modeled at various levels of detail. This also shows how to represent state-based models of computation using the SPI model.

Consider the finite state machine (FSM) in Figure 6.5 having four states ($A, B, C, D$). The transitions between the states are characterized by a label $e_i/(lat, e_o)$ where $e_i \in \{a, b, s\}$ denotes an event ($*$ denotes the wildcard) triggering the transition and the tuple $(lat, e_o)$ provides an abstraction of the action associated with the transition with respect to its latency $lat$ and its produced output event $e_o \in \{u, d\}$ ($\emptyset$ denotes that no event is produced). This abstraction of the actions corresponds to the SPI abstraction in terms of being restricted to external behavior while functional details are neglected as they would be abstracted in a corresponding SPI representation anyway.

In the following, this FSM is abstracted and represented by a SPI process $P_{fsm}$ at various levels of details. This process is embedded in a SPI process network depicted in

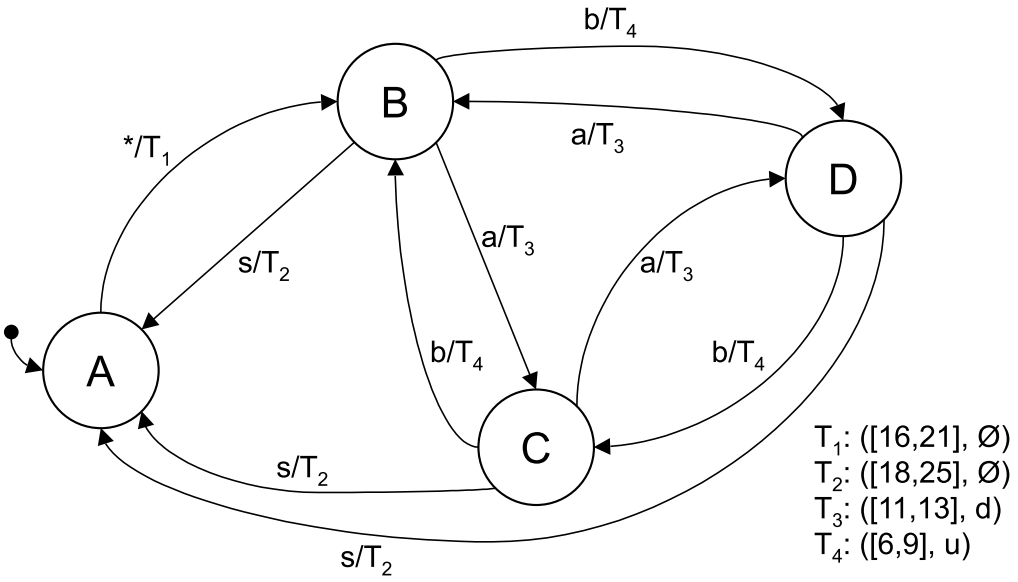



Figure 6.5: Example finite state machine

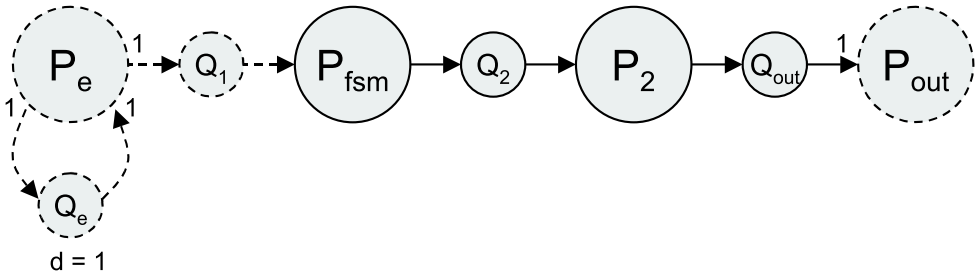



Figure 6.6: Process network with process $P_{fsm}$ abstracting the finite state machine depicted in Figure 6.5

Figure 6.6 where process $P_e$ produces events for the FSM with a minimum inter-arrival time of 30 time units as denoted by the latency constraint $LC_{\{P_e \to Q_e \to P_e\}} = [30, \infty]$. The events produced by the FSM activate process $P_2$ performing different computations depending on the type of the produced event. This is captured by $P_2$ having the following two modes:

$$m_i = (r_{Q_2}, s_{Q_{out}}, lat_{P_2})$$
$$m_1 = (1, 1, [16, 20])$$
$$m_2 = (1, 1, [9, 14])$$

All abstractions of the finite state machine are based on the modeling assumption that one process execution represents the reaction of the state machine to a single event. Furthermore, all input events arrive through channel $Q_1$ while all output events are produced on channel $Q_2$. Assuming a single processor implementation, the worst case overall computational load of this processor is calculated for the different representations in order to evaluate their accuracy.

The first representation of the FSM has a single behavior with an input data rate $r_{Q_1} = 1$, an output data rate interval $s_{Q_2} = [0, 1]$, and a latency interval $lat_{P_{fsm}} = [6, 25]$. While the fixed input data rate of 1 follows directly from the above modeling assumption (process execution models reaction to single event), the output data rate interval denotes the uncertainty whether a transition with or without event production has been taken. The latency interval is calculated by simply finding the smallest and largest latency values of all transitions. While having full knowledge of the activation of process $P_{fsm}$, the abstracted transition behavior leads to the worst case behavior in terms of maximum inferred computational load of executing for 25 time units and producing 1 output event at each process execution. Together with process $P_2$ this yields a worst case overall computational load of

$$\frac{lat_{P_{fsm},max} + lat_{P_2,max}}{t_{arrival,min}} = \frac{25 + 20}{30} = 1.5$$

In order to improve the analysis accuracy, process modes are introduced to represent the different transitions of the state machine. Then, process $P_{fsm}$ has the following four process modes:

$$m_i = (r_{Q_1}, s_{Q_2}, lat_{P_{fsm}})$$

$$m_1 = (1, 0, [16, 21])$$

$$m_2 = (1, 0, [18, 25])$$

$$m_3 = (1, 1, [11, 13])$$

$$m_4 = (1, 1, [6, 9])$$

This representation reveals that for the first two process modes with a long latency no output events triggering additional computations by $P_2$ are produced. Thus, the worst case overall computational load can be reduced to

$$\frac{\max(lat_{P_{fsm}(m_1,m_2)}, lat_{P_{fsm}(m_3,m_4)} + lat_{P_2})}{t_{arrival,min}} = \frac{\max(25, 13 + 20)}{30} \approx 1.09$$

A further step to a more accurate representation of the finite state machine is to model the events not only by their amount but also by their type. This is done by introducing a mode tag $event$ with a domain $D(event) = \{'a', 'b', 's', 'u', 'd'\}$ and updating the modes $m_3$ and $m_4$ as follows

$$m_3 = (1, 1, [11, 13], \{'true' \mapsto def(Q_2.tag(1), event, \{'d'\})\})$$

$$m_4 = (1, 1, [6, 9], \{'true' \mapsto def(Q_2.tag(1), event, \{'u'\})\})$$

in order to specify the produced event type. Accordingly, the activation function of process $P_2$ has to be adapted to evaluate the mode tag set of the incoming tokens:

$$(Q_2 \geq 1) \wedge (event, \{'u'\}) \in Q_2.tag(1) \mapsto \{m_1\}$$

$$(Q_2 \geq 1) \wedge (event, \{'d'\}) \in Q_2.tag(1) \mapsto \{m_2\}$$

This visualizes the existing correspondence between the events produced by the FSM and the modes of process $P_2$ and helps to further reduce the worst case overall computational load to

$$\frac{\max(\ldots, lat_{P_{fsm}(m_3)} + lat_{P_2(m_2)}, lat_{P_{fsm}(m_4)} + lat_{P_2(m_1)})}{t_{arrival,min}} = \frac{\max(25, 27, 29)}{30} \approx 0.97$$

the state of a SPI model is only composed of the channel contents (amounts of tokens and mode tags)

The last representation contains every detail of the FSM but the state itself. Noting that the state of a SPI model is only composed of the channel contents (numbers of tokens and mode tags), the state of a process can be seen as input data coming from the previous execution of the same process and thus can be modeled by introducing a feedback queue starting and ending in this process. This queue is virtual as it does not have to be implemented since the local process data is already accounted for by the process parameter $mem_{p,stat}$ (see Section 4.3.2.3). The state of the process then can be encoded in a mode tag and thus made available for the activation function. For process $P_{fsm}$ in particular, this results in the feedback queue $Q_s$ already depicted in Figure 6.6 and assuming a state encoding mode tag $state$ with $D(state) = \{'A', 'B', 'C', 'D'\}$ in the following activation rules capturing the outgoing transitions of e. g. state $B$:

$$(Q_s \geq 1) \wedge (state, \{'B'\}) \in Q_s.tag(1) \wedge (Q_1 \geq 1) \wedge (event, \{'a'\}) \in Q_1.tag(1) \mapsto \{m_3\}$$

$$(Q_s \geq 1) \wedge (state, \{'B'\}) \in Q_s.tag(1) \wedge (Q_1 \geq 1) \wedge (event, \{'b'\}) \in Q_1.tag(1) \mapsto \{m_4\}$$

$$(Q_s \geq 1) \wedge (state, \{'B'\}) \in Q_s.tag(1) \wedge (Q_1 \geq 1) \wedge (event, \{'s'\}) \in Q_1.tag(1) \mapsto \{m_2\}$$

However, as the outgoing transitions of states $B$, $C$, and $D$ are equivalent with respect to their external behavior, these states can be merged on the SPI level as possible functional differences of their associated actions are not represented. Then, assuming a state encoding mode tag $state$ with $D(state) = \{'A', 'BCD'\}$, the most accurate and compact representation of the finite state machine on the SPI level is characterized by the process modes

$$m_i = (r_{Q_1}, s_{Q_2}, lat_{P_{fsm}}, TP_{P_{fsm}})$$

$$m_1 = (1, 0, [16, 21], \{tp_{BCD}\})$$

$$m_2 = (1, 0, [18, 25], \{'true' \mapsto def(Q_s.tag(1), state, \{'A'\})\})$$

$$m_3 = (1, 1, [11, 13], \{'true' \mapsto def(Q_2.tag(1), event, \{'d'\}), tp_{BCD}\})$$

$$m_4 = (1, 1, [6, 9], \{'true' \mapsto def(Q_2.tag(1), event, \{'u'\}), tp_{BCD}\})$$

$$\text{with} \quad tp_{BCD} : 'true' \mapsto def(Q_s.tag(1), state, \{'BCD'\})$$

and an activation function consisting of the following rules:

$$(Q_s \geq 1) \wedge (state, \{'A'\}) \in Q_s.tag(1) \wedge (Q_1 \geq 1) \mapsto \{m_1\}$$

$$(Q_s \geq 1) \wedge (state, \{'BCD'\}) \in Q_s.tag(1) \wedge (Q_1 \geq 1) \wedge (event, \{'a'\}) \in Q_1.tag(1) \mapsto \{m_3\}$$

$$(Q_s \geq 1) \wedge (state, \{'BCD'\}) \in Q_s.tag(1) \wedge (Q_1 \geq 1) \wedge (event, \{'b'\}) \in Q_1.tag(1) \mapsto \{m_4\}$$

$$(Q_s \geq 1) \wedge (state, \{'BCD'\}) \in Q_s.tag(1) \wedge (Q_1 \geq 1) \wedge (event, \{'s'\}) \in Q_1.tag(1) \mapsto \{m_2\}$$

However, the representation of state does not change the worst case overall computational load for this example. But concerning the external behavior (timing and event consumption and production), process $P_{fsm}$ exactly represents the FSM. This example shows that the SPI model can be used to model a finite state machine at various levels of accuracy. Additionally, it also shows that an increase in modeling accuracy does not always improve the analysis results. In this example, this is due to the particular interaction of processes $P_{fsm}$ and $P_2$.

Based on the example a general mapping rule can be formulated for the representation of a finite state machine by a SPI process.

1. Generate a process $P$ with a feedback queue $Q_s$. Encode the $n_s$ states of the FSM by a mode tag $state$ with $|D(state)| = n_s$ to be communicated on $Q_s$. Possibly consider a state minimization in order to merge states with equivalent or similar external behaviors as functional differences are not captured due to abstraction (compare states $B$, $C$, and $D$ in above example).

2. For each transition of the FSM, capture its associated actions by a process mode including a data token written to $Q_s$ carrying a mode tag identifying the next state.

3. For each transition, generate an activation rule, based on the outgoing state (mode tag on $Q_s$) and triggering events (input queues) of this transition, which maps to the process mode representing this transition.

This rule can be extended in order to include also hierarchical finite state machines such as StateCharts [42, 41]. But additional analysis is necessary to capture the dependencies caused by internal signals and variables. Another possibility is to represent the state machine not by a single SPI process but rather by one process per state. Then, such a process has a mode for each outgoing transition and produces a token to activate the process representing the next state. However, some additional processes are necessary e. g. to synchronize internal variables or output events generated by several different states. This supports distributed implementation of a state machine and is presented in a rudimentary form in [80].

An example for a more complex state-based model of computation is the FunState model ([88] or Section 3.3.3). In the following, it is shown how a FunState component, consisting of a finite state machine and a data flow network controlled by the FSM, can be represented by a SPI process. The main difference to the previous approach is that the state of a FunState component is composed of not only the state of its state machine but also the content of its internal storage elements (registers and FIFO queues). As the queues are unbounded, this results in an infinite state space that cannot be visualized using a single feedback channel since there is only a finite mode tag set to encode the state. Thus, one virtual channel is used for encoding the states of the FunState component's state machine using mode tags. Additionally, for each internal storage element that is contained in a predicate of the component's state machine, a virtual feedback channel is added to the corresponding SPI process. Then, each transition in the FunState component's state machine can be represented by a mode of the SPI process. The behavior and activation rules of this mode can be directly derived from the triggered actions and the predicates respectively.

In Figure 6.7, an example FunState component $C$ and its corresponding SPI process $P$ are shown. The predicates of the transitions of the component's state machine are as follows:

$$\begin{aligned} p_1: &\quad (i_1.num \geq 1) \\ p_2: &\quad (q_1.num \geq 1) \wedge (i_2.num = 0) \\ p_3: &\quad (q_2.num \geq 1) \\ p_4: &\quad (q_1.num \geq 1) \wedge (i_2.num \geq 1) \\ p_5: &\quad (i_1.num \geq 1) \wedge (q_2.num = 0) \end{aligned}$$

In the SPI model, the states of the state machine of the FunState component $C$ are encoded by a mode tag $state$ with $D(state) = \{'s_1', 's_2', 's_3', 's_4'\}$ on the virtual feedback channel $q_s$. The other feedback channels $q_1$ and $q_2$ represent the corresponding internal queues of $C$. Note that register $r$ is not represented since it is not featured in a transition predicate of $C$'s state machine. Then, the process $P$ can be modeled with a mode set $M = \{m_1, m_2, m_3, m_4\}$ and

$$m_i = (r_{i_1}, r_{i_2}, r_{q_s}, r_{q_1}, r_{q_2}, s_o, s_{q_s}, s_{q_1}, s_{q_2}, TP)$$

$$\begin{aligned} m_1 &= (1, 0, 1, 0, 0, 0, 1, 1, 0, 'true' \mapsto def(q_s(1), state, \{'s_2'\})) \\ m_2 &= (0, 0, 1, 1, 0, 0, 1, 0, 2, 'true' \mapsto def(q_s(1), state, \{'s_3'\})) \\ m_3 &= (0, 0, 1, 0, 1, 1, 1, 0, 0, 'true' \mapsto def(q_s(1), state, \{'s_3'\})) \\ m_4 &= (0, 1, 1, 1, 0, 1, 1, 0, 0, 'true' \mapsto def(q_s(1), state, \{'s_4'\})) \end{aligned}$$

and an activation function consisting of the following four rules:

$$\begin{aligned} &(p_1 \wedge ((state, \{'s_1'\}) \in q_s.tag(1))) \vee \\ &(p_1 \wedge ((state, \{'s_4'\}) \in q_s.tag(1))) \vee \\ &(p_5 \wedge ((state, \{'s_3'\}) \in q_s.tag(1))) \quad \mapsto \{m_1\} \end{aligned}$$

$$p_2 \wedge ((state, \{'s_2'\}) \in q_s.tag(1)) \mapsto \{m_2\}$$

$$p_3 \wedge ((state, \{'s_3'\}) \in q_s.tag(1)) \mapsto \{m_3\}$$

$$p_4 \wedge ((state, \{'s_2'\}) \in q_s.tag(1)) \mapsto \{m_4\}$$

In this representation, mode $m_i$ of process $P$ is equivalent to the function $f_i$ in component $C$. Thus, e. g. the execution of $P$ in mode $m_2$ has the same effect as the execution of function $f_2$ as they both consume a token from $q_1$ and produce two tokens on $q_2$. By producing a token with an associated mode tag $(state, \{'s_3'\})$ on channel $q_s$, the transition of $C$'s state machine to state $s_3$ is modeled.

In summary, state-based modeling using the SPI model may not be as intuitive as in graphical FSM-based notations, although the recently presented Abstract Codesign Finite State Machines model (ACFSM) [85] specifies state machines in a very similar state-transition-form. However, the SPI representation allows a seamless integration with other models of computations and the representation at various levels of abstraction helps to reduce analysis complexity.

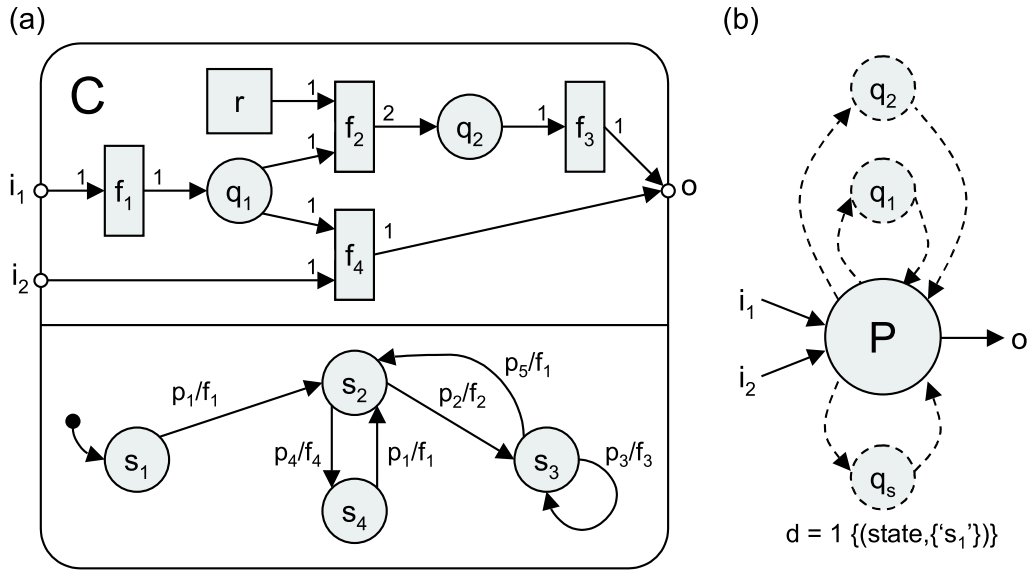



Figure 6.7: A FunState component (a) and its corresponding SPI process (b)

## 6.2 Mapping and Transformation

This section deals with certain aspects of the transformation of input languages to the SPI model and gives examples concerning this step. In particular, the extraction of implementation-dependent parameters, a transformation approach for a simulation model, and an example for designer choices during transformation are presented.

### 6.2.1 Parameter Extraction

An important property of the SPI parameter intervals is that the bounds on the parameter value have to be conservative in order to prevent a false acceptance of a system implementation leading to a violation of system constraints. Simulative determination of non-functional properties even of a single process can not guarantee to yield conservative bounds as the test pattern selection for the coverage of the extreme cases is in general undecidable and exhaustive simulation is too costly. Thus, the extraction of the SPI parameter intervals from the input languages is based on a formal analysis tool suite called SYMTA (SYMbolic Timing Analysis) [95] that other than suggested by its name is not only capable to determine the parameter intervals for latency times but also for power consumption and communicated data.

The SYMTA tool suite extends the well-established sum-of-basic-blocks method [70], a formal static software execution cost analysis approach where the overall process execution cost is the sum of all basic block execution costs multiplied by the corresponding execution count for each of the basic blocks. The added value of the SYMTA approach is an automated path analysis step which increases the accuracy (narrower intervals). Thus, the SYMTA analysis approach consists of the following three steps:

- **Identification of process segments.** Process segments are sequences of basic blocks. If possible, execution paths having a single input data independent control flow

across basic block boundaries are identified and the corresponding basic blocks are clustered to process segments.

- **Determination of parameter intervals for each process segment.** For all process segments with input-data independent control flow, simulation or execution profiling (combined with added conservative overhead considering unknown pipeline or cache states as well as data-dependent instruction cost) can be used to determine conservative parameter intervals regardless of the used test patterns. Due to the raised granularity, the number of points where the overheads have to be considered is less than for the standard approach [70] leading to a higher analysis accuracy.

- **Combination of process segment parameter intervals.** Based on structural and functional constraints two ILP (Integer Linear Programming) problems are formulated and solved yielding conservative bounds on the execution counts of each process segment. Then, the overall parameter intervals of the complete process are determined by an addition of the process segment parameter intervals multiplied with their respective execution counts.

A more formal presentation of the analysis approach including formal definitions of process segments and their classification can be found in [95].

Since SYMTA internally is based on syntax and control flow graphs, it is not restricted to a specific input language. Currently, one front end (translation from input language to internal data structures) has been implemented for $C^x$ [26], a C derivative that extends ANSI-C by generic `send` and `receive` functions which implement the inter-process communication. However, SYMTA can be easily enhanced to cover more recent system-level languages like SpecC [31] and SystemC [47]. This flexibility facilitates the use of SYMTA for the determination of implementation-dependent parameters of SPI processes originating from a wide variety of specification languages as they typically provide a possibility to automatically generate code in a C derivative. This has already been demonstrated for Simulink and the Real-Time Workshop code generator ([57] and Section 6.2.2). As the back end (process segment parameter interval determination) is equally flexible allowing to use off-the-shelf simulators as well as evaluation platforms, the implementation-dependent parameters latency and power consumption can be determined for a wide variety of target architectures. Even an extension to hardware component analysis is possible.

So far, only the determination of parameter intervals for the complete process has been discussed. However, SYMTA supports the specification of an execution context selecting a set of execution paths for analysis yielding parameter intervals for the corresponding SPI process mode. More details on the application of SYMTA for the determination of SPI parameters are presented in [98].

In the following, an example for the SPI parameter extraction using SYMTA is presented. The example process is an image filter operating on a packet stream containing the images. If a packet is addressed to its system component, the filter process performs an ”unlikely dot” filtering on the image data and forwards the filtered image to another process. Possible execution contexts are the processing of a ”large” or a ”small” image and address match or miss. The pseudo code of the process is given in Figure 6.8.

Tables 6.1 and 6.2 show the parameter intervals of the filter process with respect to latency time, power consumption of the processor core and communicated data for all

```
89:  header = receive(INPUT, HEADER_SIZE);
     for all pixels              /* Context: Size    */
       picture[y][x] = receive(INPUT, 1);

122: if(address == MY_ADDRESS) { /* Context: Address */
124:   for all pixels {
         for a 3*3 pixel window {
143:       if(without_center)
             average = sum/8;
           else average = sum/9;
           }
151:       if(abs(picture[y][x]-average)>threshold)
             send(OUTPUT, average, 1);
           else send(OUTPUT, picture[y][x], 1);
       }
     }
```

Figure 6.8: Pseudo code of the filter process

combinations of execution contexts. The intervals are of the form $[b_{min},\ b_{max}]$ where $b_{min}$ denotes the lower bound on the value of the respective property while $b_{max}$ denotes the upper bound. The values were obtained using the SYMTA tool for a StrongARM with 80 MHz core frequency, 40 MHz bus frequency and 25 ns memory cycle time including local cache simulation [98].

| Latency [ms] / Power [mWs] | | Address not considered | Address miss | Address match |
|---|---|---|---|---|
| Size not considered | Latency | [5, 681] | [6, 40] | [38, 681] |
| | Power | [2, 117] | [2, 9] | [21, 117] |
| Large Picture | Latency | [19, 572] | [20, 39] | [265, 572] |
| | Power | [8, 107] | [8, 9] | [97, 107] |
| Small Picture | Latency | [5, 67] | [6, 13] | [38, 64] |
| | Power | [2, 25] | [2, 4] | [21, 24] |

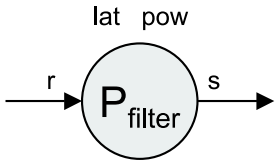
Table 6.1: Parameter intervals for latency and power consumption of the filter process

| Send Data [kB] / Receive Data [ms] | | Address not considered | Address miss | Address match |
|---|---|---|---|---|
| Size not considered | Snd | [0, 24.4] | [0, 0] | [5.9, 24.4] |
| | Rec | [6.2, 25.0] | [6.2, 25.0] | [6.2, 25.0] |
| Large Picture | Snd | [0, 24.4] | [0, 0] | [24.4, 24.4] |
| | Rec | [25.0, 25.0] | [25.0, 25.0] | [25.0, 25.0] |
| Small Picture | Snd | [0, 5.9] | [0, 0] | [5.9, 5.9] |
| | Rec | [6.2, 6.2] | [6.2, 6.2] | [6.2, 6.2] |

Table 6.2: Parameter intervals for sent and received data amounts of filter process

When comparing the values for the different combinations of contexts, it can be seen

that the utilization of specified contexts during analysis of the filter process helps to substantially narrow the extracted behavioral intervals. For the communicated amount of data, this context dependent analysis even yields deterministic values i. e. the upper and lower bounds of an interval are equal.




lat pow s r

*Values for SPI representation with single mode*

$m_{\text{single\_mode}}$ = ( [6, 572], [2, 107], [0.0, 24.4], [6.2, 25.0] )

*Values for SPI representation with four modes*

$m_{\text{small,match}}$ = ( [38, 64], [21, 24], [5.9, 5.9], [6.2, 6.2] )
$m_{\text{small,miss}}$ = ( [6, 13], [2, 4], [0, 0], [6.2, 6.2] )
$m_{\text{large,match}}$ = ( [265, 572], [97, 107], [24.4, 24.4], [25.0, 25.0] )
$m_{\text{large,miss}}$ = ( [20, 39], [8, 9], [0, 0], [25.0, 25.0] )

Figure 6.9: SPI representations of filter process

An interesting effect is that the maximum latency of 681 ms without considering any execution context (see upper left element of Table 6.1) is not contained in the behavioral intervals for neither the large nor the small image context (572 ms and 67 ms respectively). This is due to the fact, that for each of both contexts two process segments have merged such that the worst case assumptions on the cache and pipeline state for the beginning of the second segment can be dropped. Even though the intervals seem wide, the accuracy is actually high. This becomes evident when comparing the results to the standard approach [70] that yields upper bounds 6368 ms and 887 ms for a large and a small image respectively compared to 572 ms and 67 ms obtained by SYMTA.

Based on the obtained parameter intervals, a SPI representation of the filter process can be generated that may be used to analyze the overall system performance. Two SPI representations of the filter process using the obtained parameter intervals with a single mode and four modes are depicted in Figure 6.9.

### 6.2.2 Model Transformation

Key to the transformation of the heterogeneous functional specification to a homogeneous model is the representation of the input models of computation in the SPI model. Particularly interesting is the representation of models which are targeted to specification and simulation rather than to synthesis. These models are typically based on assumptions which are either non-implementable (e. g. computation or communication in zero time) or unnecessarily restrictive with respect to the design space (e. g. computation according to fixed internal timing). While these assumptions may be represented in the SPI model (e. g. zero latency enforced by latency constraint), it is not favorable to do so as the purpose of the SPI model is to analyze and optimize non-functional properties of a system

implementation. Thus, a SPI representation of such a model of computation should rather be meaningful in the context of synthesis and consider the implementation interpretations of these simulation constructs as used by code generators. In the following, this transformation principle is explained using the Simulink model of computation described in Section 2.2.2 and its transformation to SPI [57] as an example.

The Simulink model of computation consists of blocks communicating unidirectionally via signals and being executed periodically in zero time at exact time steps. The communication imposes a partial ordering of blocks being executed at the same time step (and thus theoretically at the same time). The implementation interpretation of this timing by Simulink code generators (e. g. Real-Time Workshop (RTW) [73]) is to relax the exact time steps by sequentially executing all blocks to be executed at a certain time step. The zero time execution is interpreted as a timely completion requirement where timely means before the next time step.

A more systematic implementation interpretation of the Simulink model of computation can be based on the consideration of model properties that have to be preserved in order to keep the causality of model execution. These are

- the relative execution rates of the Simulink blocks,
- the partial ordering of blocks executing at the same time step, and
- the access sequence for each signal.

By preserving these properties, the network functionality is not changed. A consequence is that the absolute and exact timing of the Simulink model of computation can be replaced by execution dependencies in the SPI model. This differs from the implementation interpretation of RTW which still relies on the time step execution. An example is that the requirement for a block execution to be completed before the next time step can be relaxed. This is due to the fact that the access sequence is preserved as long the block execution completes before the next block reading a produced signal starts execution. This read access may happen several time steps later such that the mobility interval of the writing block is increased.

Based on the example in Figure 6.10 (taken from [57]), the actual representation of Simulink in the SPI model is explained. The Simulink blocks are modeled by SPI processes communicating via registers modeling the Simulink signals. The model causality is kept by virtual queues between communicating processes as described in Section 6.1.1 where the queue in direction of the communication preserves the relative execution rate and the partial order of concurrently enabled blocks while the back queue along with a suitable number of preassigned tokens ensures the correct access sequence. This dependency-based representation completely abstracts the exact timing as enforced by Simulink and thus spans an increased design space. However, at the boundaries between system and environment the exact Simulink timing may be required (e. g. exact periodic sampling of input signal). These wanted and critical timing constraints then can be reintroduced at the SPI level. In the example, this is the case for Simulink block $B_1$ and is modeled by a latency constraint $LC_{\{P_1 \mapsto Q_s \mapsto P_1\}} = 1$ enforcing the specified sample time of 1 time unit for the corresponding SPI process $P_1$.

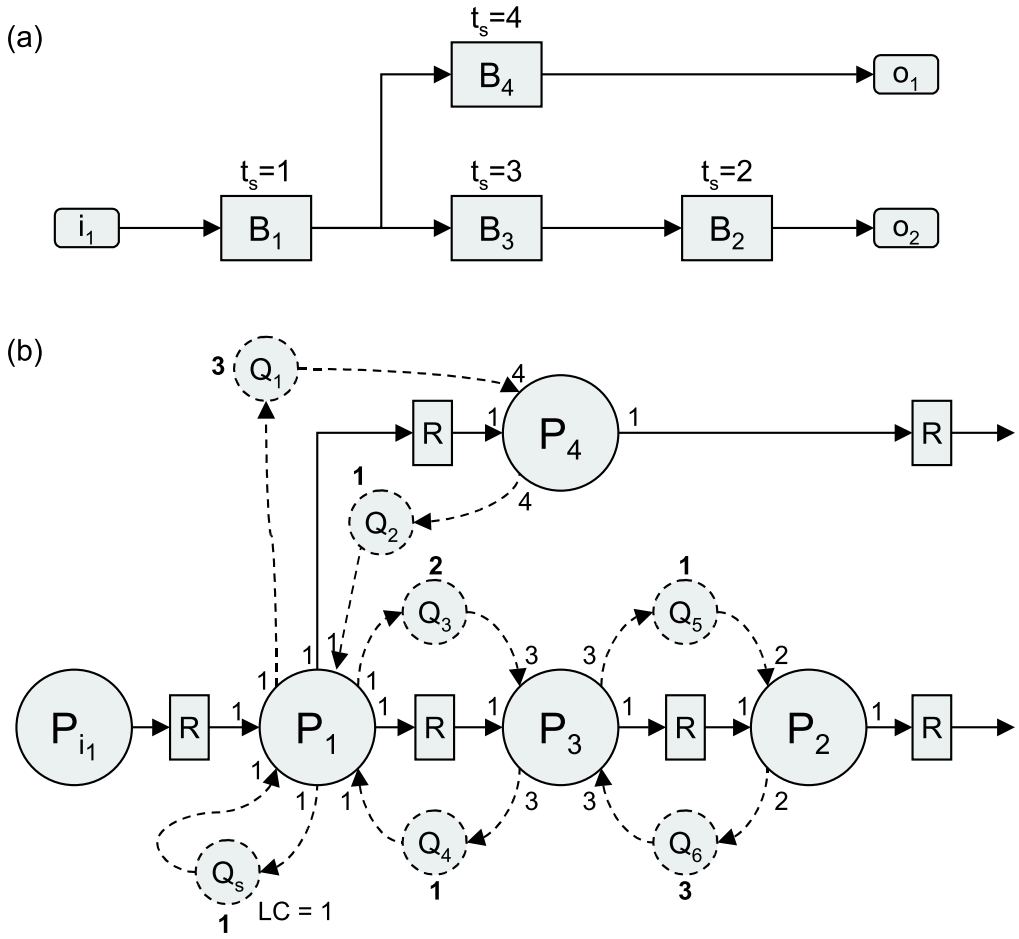



Figure 6.10: Example for the translation of a Simulink model (a) into a SPI representation (b)

More examples, formal translation rules, granularity adaptation, and details on the transformation of constructs like triggered subsystems, continuous-time blocks etc. can be found in [57] and [56].

### 6.2.3 Granularity and Modeling Accuracy

Based on an example system, the design decisions during input transformation with respect to granularity and modeling accuracy are discussed in the following. This example system is a remote motor controller which is specified as depicted in Figure 6.11. The system collects message parts from a bus and tests them for an error ($P_1$), decodes the collected message ($P_2$) and sends a control word to the motor control loop ($P_3$). In this system, process $P_1$ is specified in a state-based language, process $P_2$ in a C derivative and process $P_3$ in a synchronous data flow language. The interaction of the processes and the environment is loosely defined. $P_1$ and $P_2$ are both periodic processes that are activated every $t_1$ and $16 \cdot t_1$ time units respectively. Both processes have a deadline at the end of their period. Process $P_3$ is driven by a periodic input. There is a timing constraint that

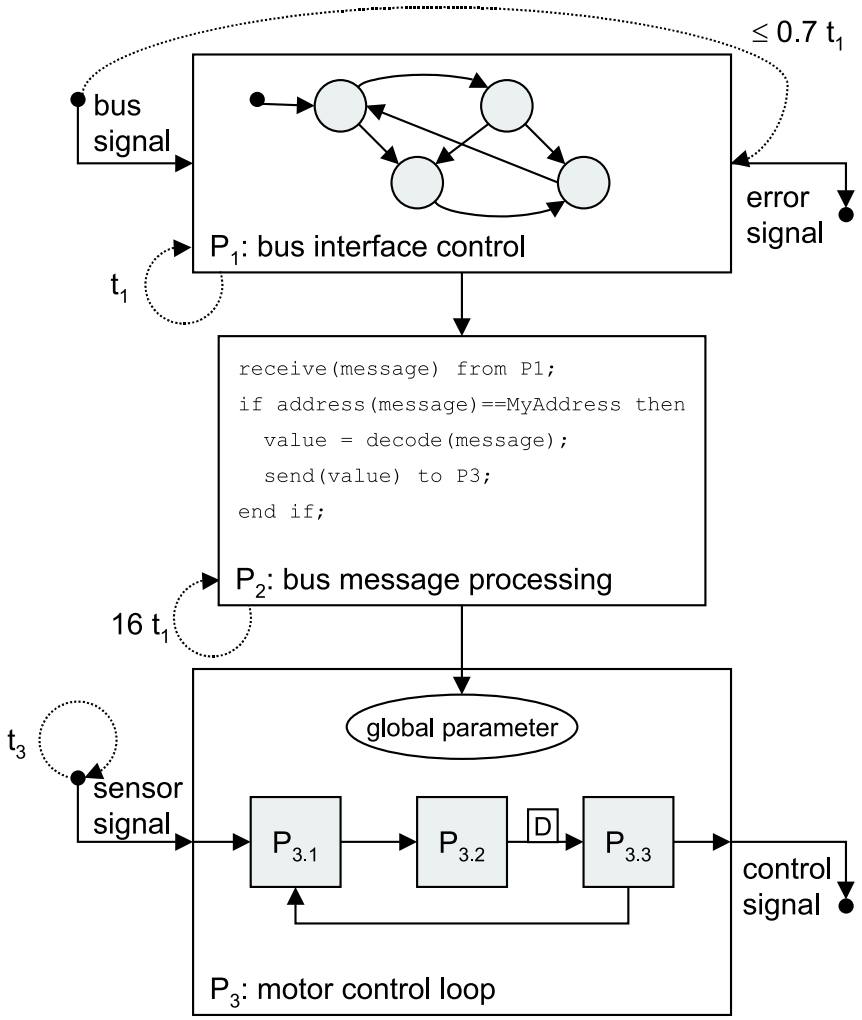



Figure 6.11: Remote motor controller

constrains the maximum response time to an erroneously received message part to be no longer than $0.7$ of the bus period $t_1$.

A first decision of the input transformation step is to specify granularity i. e. how many SPI elements are used to represent the input language elements. In the example, the particular choice is whether to map the SDF graph in process $P_3$ to a single SPI process or one SPI process per actor. This decision depends mainly on the functional complexity of the dataflow actors. If the actors consist only of a few operations each, they should probably be merged in order to reduce problem size and a possible run-time overhead for a dynamic scheduling implementation. However, in this example, the SDF graph is mapped to three SPI processes ($P_{3.1}$, $P_{3.2}$ and $P_{3.3}$) as can be seen in the SPI representation of the remote motor controller depicted in Figure 6.12.

The next decisions are on which SPI elements to use in order to represent the interaction of processes and environment and on where to draw the line between system and environment. In the example, the signal sources ($P_{bus}$ and $P_{sensor}$) and sinks ($P_{error}$ and $P_{motor}$) are represented by virtual processes but the channels between environment and system are not virtual, since this communication has to be implemented with the system

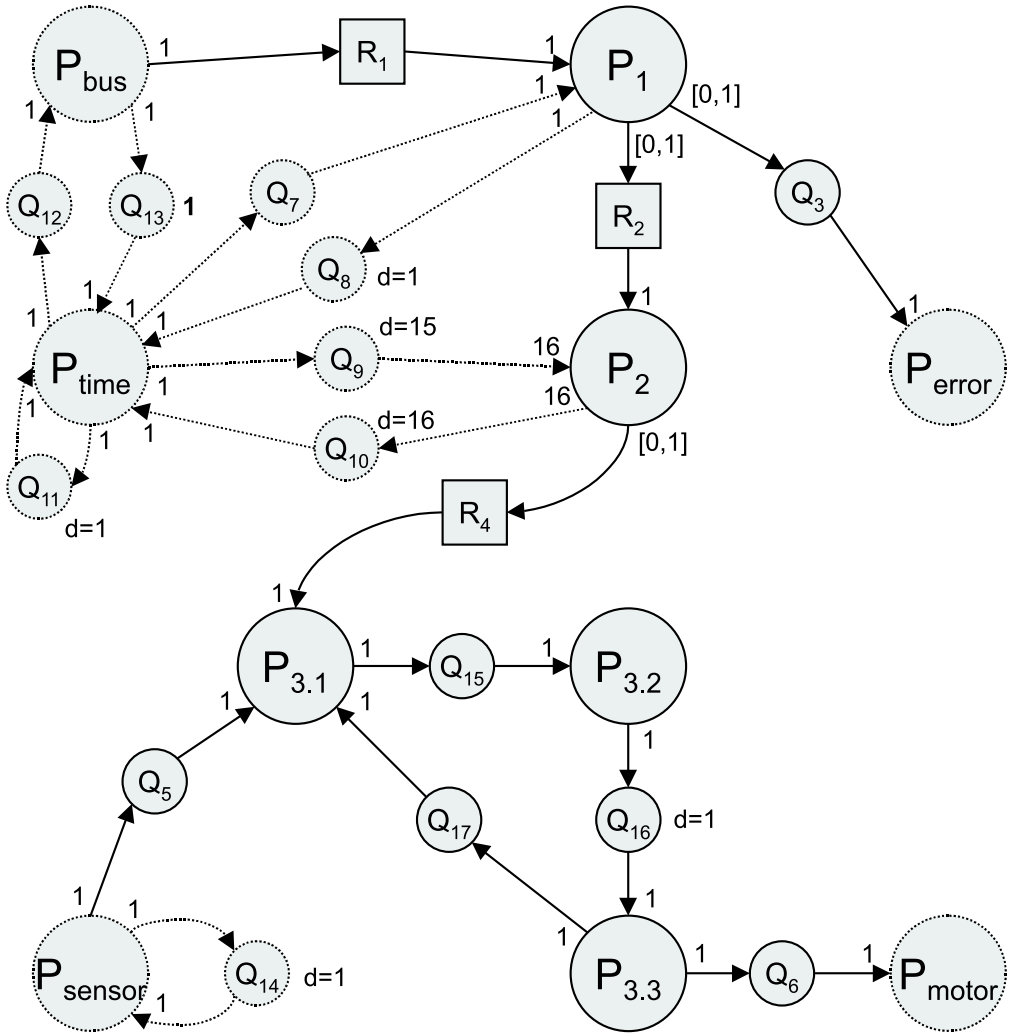



Figure 6.12: Remote motor controller (SPI representation)

(e. g. using memory mapped I/O). The input channels of processes $P_1$ and $P_2$ are modeled using registers as $P_1$ and $P_2$ are time-driven processes and will be executed whether new input data has arrived or not. Similarly, the interface between the time-driven processes and the SDF part of the design is modeled by a register as both parts are not synchronized and process $P_{3.1}$ simply reads the latest control parameters stored in $R_4$.

Instead of using for each time-driven process a separate virtual process for the timed activation with associated deadline constraint as described in Section 6.1.2, a single timer process $P_{time}$ is used that activates processes $P_1$, $P_2$, and signal source $P_{bus}$ via virtual queues denoting relative execution rates with a constrained mobility interval (see Section 6.1.1). This mechanism is explained for process $P_1$ which has to be executed once during every period $t_1$ and thus once between two executions of the virtual timer process $P_{time}$ having an exact period of $t_1$. To model this fact, two virtual channels $Q_7$ and $Q_8$ with $Q_8$ having one preassigned token are introduced. Then, the first execution of $P_{time}$ causes an activation of $P_1$ whose execution in turn leads to another activation of $P_{time}$ etc. As process $P_{time}$'s executions have to be exactly $t_1$ time units apart, this models not only

a periodic activation but due to queue $Q_8$ also a required timely completion of $P_1$.

After having discussed the different choices regarding the used SPI elements, the characterization of these elements namely its parameters and thus the modeling accuracy has to be considered. This is mainly a choice of how many process modes to use and whether to visualize a possible process state or a certain data value by a mode tag. A good example is the bus interface control process $P_1$. Assuming this process is to be modeled using two modes, the different execution paths can be grouped as follows. The process either reads an erroneous message part from the bus and outputs an error message (mode $m_1$) or it reads a correctly received message part (mode $m_2$). Then, its process modes are

$$m_i = (lat, r_{R_1}, r_{Q_7}, s_{R_2}, s_{Q_3}, s_{Q_8})$$

$$m_1 = ([0.3, 0.6], 1, 1, 0, 1, 1)$$

$$m_2 = ([0.4, 0.8], 1, 1, [0, 1], 0, 1)$$

For simplicity, process latencies are expressed as fractions or multiples of $t_1$, the period of the incoming bus signals throughout the example. Note that mode $m_2$ still contains uncertainty with respect to the communication behavior of process $P_1$ as denoted by the output data rate interval for register $R_2$. This interval represents the fact that this output is only produced if the received message is the last part of a telegram, i. e. the whole telegram is written to the decoder. However, as no computational load (process executions) depends on the occurrence of this output production, the different process behaviors are not distinguished. The same holds for the output production of process $P_2$ which depends on the result of the address matching performed by $P_2$.

The reason why process $P_1$ has to be modeled using two modes instead of having only a single behavior is the constraint limiting the response time to a transmission error that is modeled by the latency path constraint $LC_{err} = LC_{(P_{bus} \to R_1 \to P_1 \to Q_3 \to P_{error})} = [0, 0.7\,t_1]$. When looking at the single behavior of process $P_1$ which could be generated by merging the intervals of both modes and would be

$$m = (lat, r_{R_1}, r_{Q_7}, s_{R_2}, s_{Q_3}, s_{Q_8})$$

$$m = ([0.3, 0.8], 1, 1, [0, 1], [0, 1], 1)$$

it becomes evident that the specification of $P_1$ using two modes is necessary in order to be able to guarantee the satisfaction of $LC_{err}$. With a single behavior a maximum execution time $0.8\,t_1$ for $P_1$ has to be assumed which violates the maximum response time $0.7\,t_1$ specified by $LC_{err}$. However, when using the two mode representation, only the maximum latency of mode $m_1$ has to be considered which is the mode that outputs the error message such that $0.6\,t_1 < 0.7\,t_1$ and the satisfaction of $LC_{err}$ becomes possible.

## 6.3 Analysis and Optimization

In this section, exemplary applications of analysis and optimization methods to SPI representations with a focus on timing are presented. In the first subsection, a static scheduling approach and a dynamic scheduling and analysis method restricted to independent processes are applied to the remote motor controller example of the previous section. This

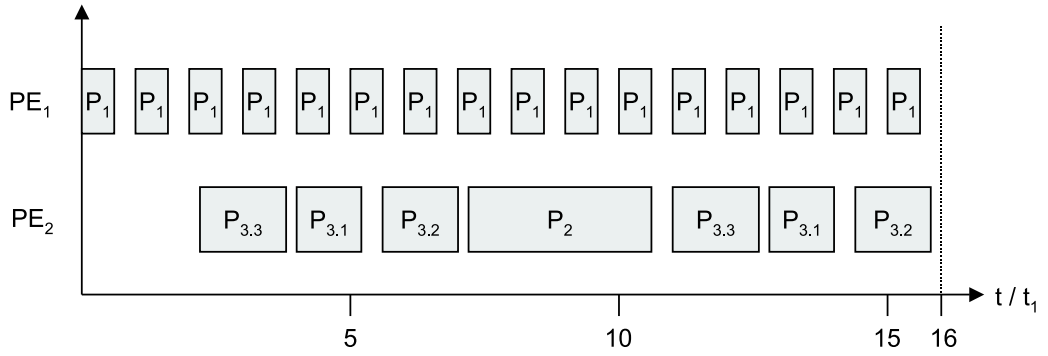



Figure 6.13: Gantt diagram representing a statically scheduled macro period ($8\,t_1$) of the remote motor controller

is followed by the presentation of a dynamic scheduling approach for SPI processes with predictable data rates and an outline on how to adapt a dynamic scheduling and analysis methodology to SPI processes with uncertain data rates.

### 6.3.1 Existing Scheduling Approaches

Based on the remote motor controller example presented in Section 6.2.3, the application of two existing scheduling approaches, static macro period scheduling (e. g. [3]) and generalized rate monotonic scheduling (GRMS) [86]) to SPI representations is shown. As both approaches assume certain process network properties in order to be applicable, a prerequisite for their application is that the SPI representations to be scheduled have these properties.

*Static macro period scheduling* assumes that the execution periods of each process are fixed and known. Given these process periods, a macro period which is the least common multiple (LCM) of all process periods can be calculated. Then, a static schedule can be generated for all process instances in the macro period by a standard approach such as [3]. For the application of this approach to the SPI representation of the remote motor controller as depicted in Figure 6.12, the process periods $T_p$ can be derived from the SPI representation. This is trivial for processes $P_1$ and $P_2$ which are specified as periodic processes (e. g. $T_{P_1} = t_1$) but could also be derived from the SPI representation (e. g. $T_{P_2} = \frac{r_{Q_9}}{s_{Q_9}} T_{P_{time}} = 16\,t_1$). Similarly, the periods of the SDF processes $P_{3.1}$, $P_{3.2}$, and $P_{3.3}$ depend on the period of the signal source driving them modeled by process $P_{sensor}$. Due to the uniform data rates throughout the SDF part their periods evaluate to $T_{P_{3.i}} = T_{P_{sensor}} = t_3$.

Furthermore, the queues between the SDF processes form precedence constraints enforcing a partial order on the execution of process instances which has to be regarded by the scheduling algorithm. These precedence constraints can be formulated as $P^k_{3.1} \mapsto P^k_{3.2}$ denoting that the $k$th execution of $P_{3.1}$ has to be completed before $P_{3.2}$ can be executed for the $k$th time. Similarly holds $P^k_{3.3} \mapsto P^k_{3.1}$ and $P^{(k+1)}_{3.2} \mapsto P^k_{3.3}$ where the instance offset $(k+1)$ originates from the preassigned token on queue $Q_{16}$.

Assuming a fixed relation $t_3 = 8\,t_1$, the macro period for the SPI representation of the remote motor controller evaluates to $T_{macro} = 16\,t_1$. A Gantt chart representing a macro

period of a possible scheduling of the motor controller on two processing elements is shown in Figure 6.13. Since the used scheduling algorithm does not support preemption $P_1$ needs to be executed on a separate processing element in order not to violate its rate constraint.

A classical scheduling approach supporting preemption is *rate monotonic scheduling (RMS)* [71], a fixed priority scheduling approach for periodically released processes where process priorities are assigned based on the process execution rate (highest rate = highest priority). Process response times and the schedulability of accordingly scheduled systems can be determined by the so called rate monotonic analysis which assumes networks with independent processes. In this context, independence means that there are no additional execution dependencies of process instances besides those defined by the process release rates. While RMS assumes for each process a deadline equal to its period, generalized rate monotonic scheduling (GRMS) and analysis also considers deadlines smaller than the period and assigns deadlines no longer according to process rates but rather according to deadlines (shortest deadline = highest priority).

For the application of generalized rate monotonic scheduling and analysis to the remote motor controller, the satisfaction of the above requirements has to be checked. While the registers $R_1$, $R_2$, and $R_4$ do not impose any execution dependencies as their access is non-blocking and the virtual queues $Q_5$, $Q_7$, and $Q_9$ model the periodic activation (release) of their reading processes, the queues $Q_15$, $Q_16$, and $Q_17$ block the execution of their reading process if they do not contain at least one token. In order to avoid these execution dependencies and allow the application of GRMS, the SDF processes are clustered i. e. represented by a single process $P_3$ as depicted in Figure 6.14.

Assuming a single processor implementation, the latency intervals and GRMS parameters of the three SPI processes are depicted in Table 6.3. Note that the latency time intervals have changed in comparison to the implementation using two processing elements as the processes are mapped to another faster processor. The GRMS parameter $C_i$ denotes the maximum execution time of process $P_i$ and can be obtained directly from the upper bound of the process latency interval. The process periods $T_i$ are derived as described for the macro period scheduling approach. While the latency path constraint $LC_{err}$ over process $P_1$ is accounted for by the process deadline $D_1 = 0.7\,t_1$, the deadlines of the other processes are assumed to be equal to their periods ($D_i = T_i$) which is as late as possible with GRMS and thus imposes the least burden on the implementation.

| Process | Latency | $C_i$ | $T_i$ | $D_i$ |
|---|---|---|---|---|
| $P_1$ | $[0.2, 0.3]\,t_1$ | $0.3\,t_1$ | $t_1$ | $0.7\,t_1$ |
| $P_2$ | $[0.6, 1]\,t_1$ | $t_1$ | $16\,t_1$ | $16\,t_1$ |
| $P_3$ | $[1.1, 1.2]\,t_1$ | $1.2\,t_1$ | $8\,t_1$ | $8\,t_1$ |

Table 6.3: Latency Time Intervals and GRMS Parameters

Then, the schedulability of the system on this processor can be validated according to generalized rate monotonic analysis by checking the satisfaction of

$$\forall i, 1 \leq i \leq n, \frac{C_1}{T_1} + \cdots + \frac{(C_i + E_i)}{T_i} \leq i(2^{1/i} - 1)$$

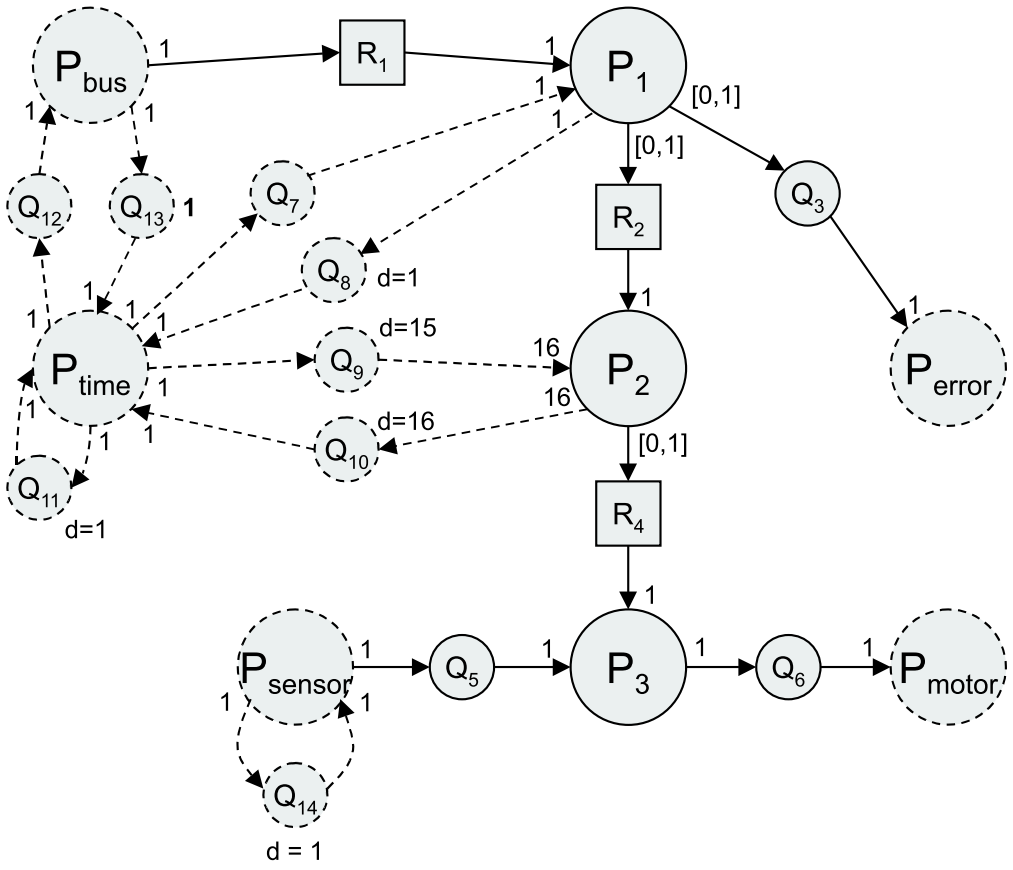



Figure 6.14: Remote motor controller (SPI representation with processes $P_{3.1}$, $P_{3.2}$, and $P_{3.3}$ being clustered as process $P_3$)

where $E_i = T_i - D_i$ and processes are ordered according to $D_i \leq D_{i+1}$. For the remote motor controller example this evaluates as follows

$$\frac{(C_1 + T_1 - D_1)}{T_1} = 0.3 + 1 - 0.7 = 0.4 \leq 1$$

$$\frac{C_1}{T_1} + \frac{C_3}{T_3} = 0.3 + 1.2/8 = 0.45 \leq 0.83$$

$$\frac{C_1}{T_1} + \frac{C_3}{T_3} + \frac{C_2}{T_2} = 0.3 + 1.2/8 + 1/16 = 0.51 \leq 0.78$$

such that the system is schedulable on the assumed processor. The latency constraint $LC_{err}$ does not have to be checked separately as the schedulability analysis has already shown that process $P_1$ never misses its deadline such that $LC_{err}$ is always satisfied.

### 6.3.2 Dynamic Response Time Optimization

Both scheduling approaches described in the previous section assume exact process periods. In many important embedded applications however, input data arrive in a non-periodic sequence. Prominent examples are burst modes and packet based transmission,

such as in automotive control engineering or Internet-based multimedia applications. In both cases, the transport medium lacks real-time characteristics to guarantee exact timing. Under these circumstances, a static order of process executions as generated by the static macro period scheduling cannot provide optimal solutions [97]. Rate monotonic scheduling and analysis allows a dynamic process ordering. However, RMS is typically not applicable to the mentioned application domains as multi-rate process communication via FIFO queues and data-driven execution is common.

Thus, a novel dynamic scheduling method has been developed which optimizes the system response time in the presence of non-deterministic timing of the system environment (input jitter). The basic idea of this method is to analyze the data dependencies of a SPI representation and derive deadlines for its processes such that 'Earliest Deadline First' (EDF) scheduling [71], a standard real-time scheduling method, of the system yields an optimal system response time. Note that the input jitter has no direct influence on the proposed scheduling method but on the achieved gain compared to existing static methods, since zero jitter stands for complete knowledge of the environment for which static methods yield optimal results as well.

In its original formulation, EDF scheduling, like RMS, is applicable to systems with independent processes only. However, Blazewicz [6] proposed an optimal method how to adjust deadlines for dependent processes such that general precedence constraints (possibly due to data-dependencies) are encoded in the revised deadlines. Then, these processes can be scheduled using the simple EDF policy such that the overall system response time is minimized. According to Blazewicz, the deadlines $d_i$ have to be revised as follows:

$$d_i^* = \min\{d_i, \min(d_j^* - lat_{P_j,max}\ ;\ P_i \rightarrow P_j)\}$$

starting from tasks with no successor and processing step by step all tasks with their successors already been processed.

In other words, deadlines of output processes are propagated through the process network to the input processes. This is done by adjusting the deadline of a process $P_i$, assuming a process $P_j$ depends on $P_i$, such that the deadline $d_i \leq d_j - lat_{P_j,max}$ ensures that process $P_i$ completes its execution at least one maximum latency $lat_{P_j,max}$ before the deadline $d_j$ of process $P_j$. If for a process no explicit deadline is specified, this deadline can be arbitrarily chosen (typical selection: duration of a macro period). This is due to the fact that not the absolute value of its deadline is important for the priority assignment of a process but the relative value in comparison with the deadlines of the other processes. These relative deadlines are obtained by performing Blazewicz's method.

Blazewicz's algorithm is targeted for process networks with general precedence constraints. However, multi-rate data dependencies as they are common in SPI representations are not covered. Furthermore, the formulation of the deadline revision procedure does not allow cycles, also a common feature of SPI process networks. Thus, the novel scheduling method generalizes Blazewicz's deadline revision such that the treatment of multi-rate data dependencies and cycles in the process network structure becomes possible.

The basic idea of the extension of Blazewicz's method to cover multi-rate data dependencies is that for each multi-rate graph there exists an equivalent single-rate graph that can be constructed by unfolding the graph [5] for a macro period. A macro period consists of a minimal set of process instances which executions transfer the system back to

its initial state. The constructed single-rate graph has a node for each instance of an actor in the macro period and single-rate data dependencies between them. Since there are only single-rate dependencies, performing Blazewicz's method on this unfolded graph yields optimal deadlines for each actor instance. Thus, optimality can still be guaranteed for graphs with multi-rate data dependencies. To avoid the possibly computationally intensive explicit construction of this unfolded graph, we extend Blazewicz's deadline revision formula to cover all actor instances in a macro period. This can be divided into finding the number of instances of each process per macro period (based on [67]) and determining the dependent process instances for each multi-rate data dependency.

A single-rate data dependency from process $P_i$ to process $P_j$ directly defines that the $k$th instance of $P_j$ has to be preceded by the $k$th instance of $P_i$. $d_{ij}$ delays on the queue representing this data dependency result in a dependency between the $(k+d_{ij})$th instance of $P_j$ and the $k$th instance of $P_i$ (in the following denoted as $P_i^k$). However, for a multi-rate data dependency from process $P_i$ to process $P_j$, several instances of process $P_j$ may depend on the same instance of $P_i$ or one instance of $P_j$ may depend on several instances of process $P_i$. For the determination of these dependencies, the input and output data rates $s_{ij}$ and $r_{ij}$ as well as the number of delays $d_{ij}$ on the queue $Q_{ij}$ connecting the processes have to be considered.

For the derivation of the deadline for the $k_i$th instance of process $P_i$, the first instance of process $P_j$ that is dependent on the execution of $P_i^{k_i}$ has to be determined. Thus, it has to hold that $P_i^{k_i}$ produces the last token needed for the execution of instance $P_j^{k_j}$, i. e. the correct instance index is the smallest value of $k_j \in \mathbb{N}_0$ that satisfies the following expression

$$k_i \cdot s_{ij} + d_{ij} < (k_j + 1)\, r_{ij}$$

where the left side denotes the number of tokens produced by $P_i$ before the $k_i$th instance, including delays, and the right side denotes the number of tokens read by $P_j$ after the $k_j$th instance. Solving for $k_j$ leads to

$$k_j > \frac{k_i \cdot s_{ij} + d_{ij}}{r_{ij}} - 1$$

The smallest $k_j$ which meets this inequation is

$$k_j = \left\lfloor \frac{k_i \cdot s_{ij} + d_{ij}}{r_{ij}} \right\rfloor$$

Thus, the deadline $d_i^{k_i}$ of the $k_i$th instance of process $P_i$ depends on the $\left\lfloor \frac{k_i \cdot s_{ij} + d_{ij}}{r_{ij}} \right\rfloor$th instance of process $P_j$.

Delays on a data dependency may lead to a dependency on an instance that is part of a higher macro period. Since deadlines are only determined for instances in one macro period, the deadline of a higher macro period instance evaluates to

$$d_j^k = d_j^{k-q_j} + \left\lfloor \frac{k}{q_j} \right\rfloor \cdot \bar{T}_{\text{macro}} \quad \text{for } k \geq q_i$$

Additionally, the precedence constraints between instances of the same process have to be considered. This results in the following expression

$$d_i^k \leq d_i^{k+1} - \text{lat}_{P_i}$$

$$d_i^k = \min\left\{ d_i^k \ , \ d_i^{k+1} - \text{lat}_{P_i} \ , \ \min\left\{ d_j^{\left\lfloor \frac{k \cdot s_{ij} + d_{ij}}{r_{ij}} \right\rfloor} - \text{lat}_{P_j} \ ; \ P_i \rightarrow P_j \right\} \right\}$$

*starting from actors with no successors and with falling $k$.*

Figure 6.15: Modified deadline revision formula capturing multi-rate data dependencies

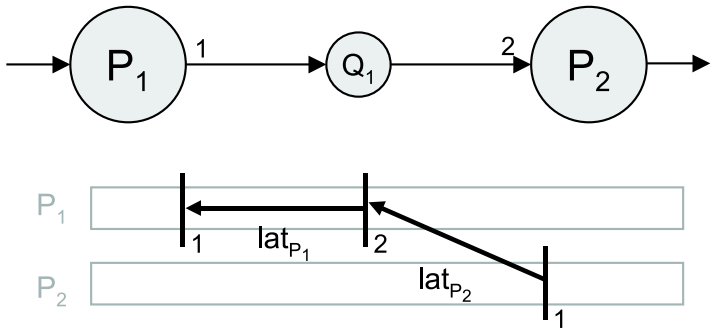



Figure 6.16: Examples for deadline revision for multi-rate data dependencies (the revised deadline of the $k$th process instance is denoted by $|_k$)

that follows Blazewicz's method for deadline revision and ensures that the $k$th instance is executed before the $(k+1)$th instance of process $P_i$.

Altogether, this results in the modified formula for deadline revision that is depicted in Fig. 6.15 and has to be computed for all process instances $P_i^k$ in one macro period starting from processes with no successors and from $k = k_{max,i}$ down to $k = 0$. This formula can be seen as a generalization of Blazewicz's formula since it yields the same results for single-rate precedence constraints and treats multi-rate constraints just as their corresponding single-rate constraints.

Figure 6.16 shows a simple example for the deadline revision for SPI processes with a multi-rate data dependency. Based on the deadline $d_2^1$ of the first instance of process $P_2$, the deadline of the second instance of $P_1$, which produces the last data token needed for the execution of process $P_2$, evaluates to $d_1^2 = d_2^1 - lat_{P_2,max}$. For the first instance of $P_1$, which has to complete before the second instance of $P_1$ may start, follows $d_1^1 = d_1^2 - lat_{P_1,max}$.

In its original formulation, Blazewicz's method does not consider cycles of precedence constraints. This is well understandable, since for precedence constraints in a single-rate graph a cycle would obviously result in a deadlock. But for precedence constraints originating from SPI data dependencies, cycles can be meaningful due to the concept of preassigned tokens which cause a data dependency to result in a precedence constraint ranging to a later instance of the receiving process. Preassigned tokens are covered by the multi-rate extension, but cycles can not be treated. This is due to the formulation of Blazewicz's algorithm that requires all deadlines of successing processes to be known for the determination of a process's deadline which is impossible for processes in a cycle.

Again, Blazewicz's idea holds also for cyclic data dependencies, the restriction is only based on the formulation of the algorithm. The solution is to use an algorithm that iteratively propagates and updates the revised deadlines according to the modified deadline revision formula until no changes occur. This algorithm is depicted below:

```
Dirty:={P_1, ..., P_N}
while Dirty≠ ∅
{
 choose P_i ∈Dirty
 {
  for all k from k_max,i downto 0
  {
   d_Temp := d_i^k
   d_i^k = ...  (formula of Fig. 6.15)
   if d_Temp ≠ d_i^k then Dirty+ =Predecessor(P_i)
   if lat_Pi > d_i^k then break DELAY_FAILURE
  }
 }
}
```

The set `Dirty` contains all processes that still have to be processed. Whenever the deadline of an actor instance is changed, all direct predecessors are added to the `Dirty` set, since the changed deadline might also effect the deadlines of their instances. This is done until the set `Dirty` is empty, i. e. no changes occurred. In case of insufficient delays in a cycle, which is a sign of an ill-formed SDF graph that would deadlock, the algorithm does not terminate, if the break condition $\mathrm{lat}_i > d_i^k$, i. e. the deadline is too early to be fulfilled, is not introduced. Together with this break condition, the algorithm is guaranteed to terminate since the deadlines monotonically decrease due to the `min`-operator.

Figure 6.17 shows an example SPI network containing a cycle and a preassigned token on queue $Q_3$. Apart from these two features, the deadlines starting from $P_2$ can be revised as in the previous example. The preassigned token on $Q_3$ causes a data dependency between the first instance of $P_3$ and the *second* instance of $P_1$. Thus, for the revised deadline of the first instance of $P_3$, the deadline of the second instance of $P_1$ (from the next macro period) has to be considered such that $d_3^1 = d_1^2 - lat_{P_1,max} = d_1^1 + T_{macro} - lat_{P_1,max}$. The data dependency between $P_2$ and $P_3$ forms the the cycle and is used to check the validity of the previously revised deadlines. In this example, the revised deadlines are valid since the deadline of $P_2$ revised according to the data dependency on $Q_2$ and shown as dotted line in Figure 6.17 is less critical than the deadline on which the revisions were based.

As the deadline revision is based on a static data flow analysis, the proposed scheduling is restricted to either constant or statically analyzable data rates (e. g. cyclo-static dataflow [5]). The consideration of non-deterministic data rates as present for the general SPI graph is problematic as the determination of a worst case dependent process instance is only solvable by complete enumeration. However, even the restriction to deterministically changing data rates allows the application of the proposed method to a large class of systems. An application example showing a 15% lower system response time for the object recognition part of an autonomous vehicle controller is presented in [97].

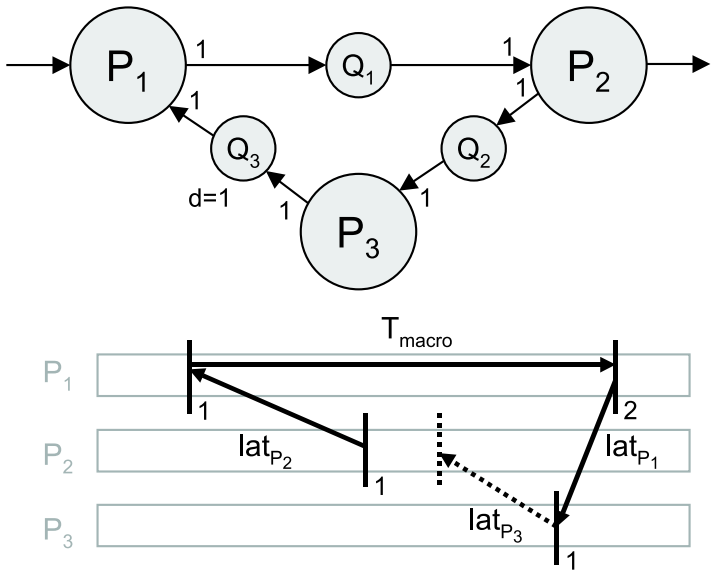



Figure 6.17: Examples for deadline revision for cyclic data dependencies (the revised deadline of the $k$th process instance is denoted by $|_k$)

### 6.3.3 Scheduling and Analysis with Uncertain Data Rates

In this section, it is shown how schedulability and end-to-end latency of a chain of SPI processes can be analyzed off-line even in the presence of uncertain data rates. A data rate is said to be uncertain if the communication behavior of a process is not off-line predictable and can only bounded by data rate intervals.

Assuming a predictable environment, static scheduling of SPI representations with fixed data rates is possible if a bounded memory schedule exists. The existence of such a schedule can be determined after [67]. Then, schedulability for a certain (set of) resources can be constructively checked by verifying that for the chosen start times no resource conflicts arise and the imposed timing constraints are satisfied. This involves the analysis of end-to-end latencies for chains of SPI processes which can be easily solved if the chain is scheduled statically. As the static schedule provides fixed process start times, the checking method for latency path constraints from Section 4.8.2 can be used to calculate the minimum and maximum end-to-end latency for the chain.

However, for SPI processes with uncertain data rates, static scheduling in general is impossible as the relative execution rates of processes varies and can not be predicted off-line. Thus, neither the macro scheduling period (see Section 6.3.1) can be determined nor the constraint checking method can be applied as the process start times are not known until run-time. Thus, dynamic process scheduling is employed. Two possible dynamic scheduling approaches are static priority driven scheduling (as presented for independent SPI processes in Section 6.3.1) and deadline driven scheduling (as presented for SPI processes with uncertain environment in Section 6.3.2).

In [51], it has been shown that for uncertain release rate characteristics of processes deadline-driven scheduling is more suitable than static priority scheduling. The reason for this is that for deadline driven scheduling, the schedulability is independent of the release

rate and solely based on the rate at which deadlines occur. As individual process deadlines can typically be chosen during implementation (see approach in previous section), schedulability can be controlled within the limits of timing constraints imposed by the environment.

In the context of the SPI model, the release rate of a process is the rate at which it is activated. For chains of SPI processes, the rate of activation of a process depends on the data rates of the preceeding processes. If these data rates are uncertain, even for an exactly known activation rate of the first process in the chain (complete environment information) the activation rate of the process is uncertain as well. Thus, for general SPI representations with communication dependencies, deadline driven scheduling seems to be more favorable than static priority driven scheduling.

For dynamic scheduling methods, task models define the system as a set of tasks and restrict task parameters such as release rates and task deadlines in order to being able to formally analyze certain real-time system properties such as schedulability. Examples are the periodic task model of [71] which assumes exactly periodic release rates with the task deadline equaling the period or the sporadic task model of [76] simply constraining the minimum time but not the maximum time between two releases of the same process. In order to decide the schedulability of a SPI representation, a suitable task model for general SPI representations has to be found.

A task model called *rate-based execution (RBE)* in which tasks are expected to execute with an *average* rate ($R = (x, y)$) of $x$ executions during $y$ time units has been proposed in [51]. This model reflects the fact that release times of processes in a process network are typically neither exactly periodic nor sporadic. Rather releases occur in an average rate based on the dataflow attribute (data rates) of the process network. Thus, RBE seems to be a suitable task model for SPI representations.

In the RBE model, tasks are assumed to have a deadline $d$ relative to their release time. This deadline however is relaxed if more than $x$ releases occur in a time interval of $y$ in order to being able to bound processor demand without explicitely bounding the release rate. Then, the possibly relaxed absolute deadline $D_i(j)$ of the $j$th instance of task $i$ released at time $t_i(j)$ is

$$D_i(j) = \begin{cases} t_i(j) + d_i & \text{if} \quad 1 \leq j \leq x_i \\ \max(t_i(j) + d_i, D_i(j - x_i) + y_i) & \text{if} \quad j \geq x_i \end{cases}$$

Furthermore, in [51] the optimality of earliest-deadline-first (EDF) scheduling for RBE task sets as well as schedulability conditions for preemptive and non-preemptive EDF scheduling of RBE task sets is shown. Additionally, in [36] and [35] a methodology for the mapping of processing graph method (PGM) graphs [65], an implementation of dataflow process networks, to the RBE task model and the subsequent schedulability and end-to-end latency analysis is presented. In the following, the main steps of this methodology are described. Afterwards suggestions are made on how this approach can be adapted in order to facilitate the analysis of schedulability and end-to-end latencies for EDF scheduling of SPI representations. For the sake of simplicity, only process chains of the form as shown in Figure 6.18 with $0 \leq i \leq n + 1$, $P_0$ being a virtual source process, and $P_{n+1}$ being a virtual sink process are considered.

For schedulability analysis, the execution rate of each process in the chain has to be determined. Assuming a rate specification $R_i = (x_i, y_i)$ for the RBE task $i$ corresponding

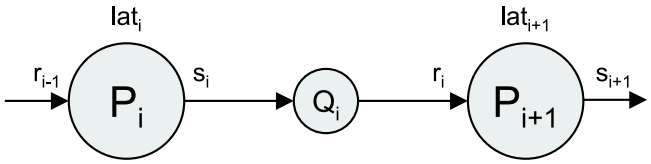



Figure 6.18: General process chain

to process $P_i$ of the chain, the process $P_i$ is executed on the average $x_i$ times during $y_i$ time units. Then after [36], the rate specification of RBE task $i+1$ corresponding to process $P_{i+1}$ is $R_{i+1} = (x_{i+1}, y_{i+1})$ where

$$x_{i+1} = \frac{s_i}{\gcd(s_i, r_i)} \cdot x_i \quad \text{and} \quad y_{i+1} = \frac{r_i}{\gcd(s_i, r_i)} \cdot y_i$$

If an RBE task set based on these maximum execution rates is schedulable, the corresponding process chain is schedulable. Note that this condition is sufficient but not necessary, as the schedulability condition of an RBE task set assumes the worst case of a simultaneous release of all $x_i$ instances of task $i$ at the beginning of an interval of length $y_i$. For dependent processes in a process network however, this worst case typically does not occur.

After the derivation of a schedulability condition, the latency analysis is covered in the following. In [35], the overall latency is broken down into two components. One of these components, the *inherent latency*, denotes the latency a token experiences if the process network is executed under the strong synchrony hypothesis, i. e. if all processes execute immediately and in zero time. After the definition of [35], the inherent latency of a process chain is the delay between the production of $s_0$ tokens on queue $Q_0$ and the next execution of the sink process $P_n + 1$. Clearly for a process chain with all processes consuming and producing a single token, the inherent latency is zero. Due to non-uniform token production and consumption however, a latency can be inferred which is a lower bound on the latency of an implementation.

Due to the strong synchrony hypothesis, implementation influences such as scheduling delays and process latencies are not considered by the inherent latency. Thus, in order to obtain the overall latency, the *imposed latency* capturing these influences has to be added to the inherent latency. This decomposition provides a very elegant separation of concerns and allows to analyze the influences of process network and implementation on the latency separately.

The inherent latency of a process chain depends on the data rate intervals of the processes, the number of tokens on the queues, and the execution rate of the source process. As the number of tokens on channels varies during execution, the latencies different tokens experience are not necessarily equal. In [35], a recursive formula is given which determines the number of additional source process executions $N_0$ required before the sink process executes for the next time. Evaluating this formula for a certain queue configuration and determining the minimum and maximum time between $(N_0 + 1)$ source process executions yields upper and lower bounds on the inherent latency. In order to obtain conservative bounds on the inherent latency valid for all possible queue configurations, $N_0$ has to be independently determined for maximum and minimum queue contents.

The imposed latency can be easily bounded. A lower bound on the imposed latency is the sum of all minimum process latencies as no causally dependent token can reach the last queue of the chain without all processes executing at least once. An upper bound on the imposed latency depends on the scheduling algorithm and its parameters, here the chosen relative deadline parameters of the RBE tasks. For a special EDF scheduling algorithm called RBE-EDF [35] which uses the overspecification due to the simultaneous release assumption of the schedulability condition in order to simplify the relative deadline assignment, the upper bound on the imposed latency evaluates to the relative deadline $d_n$ of the RBE task representing the last system process in the chain $P_n$.

Then, conservative bounds on the overall latency of a process chain implemented as an RBE task set and scheduled using the RBE-EDF scheduling algorithm can be obtained by adding the upper and lower bounds of inherent and imposed latency, respectively.

For the adaption of this methodology to SPI process chains, the following two problems have to be solved:

- The proposed methodology assumes constant data rates. SPI process chains, however, have data rate intervals. Thus, it has to be decided which data rate values have to be used in which formula in order to obtain conservative results.

- In the SPI model, the latency of a process chain is defined to be the time interval between the time a token $u$ is written to the first queue $Q_0$ of the chain and the time *all* causally dependent tokens $v$ are read from the last queue $Q_n$.[1] In [35], the latency is defined to be the time interval until the *first* causally dependent token is read. Thus, their presented algorithms have to be modified.

The separation of schedulability and latency analysis in the presented approach allow a general statement on which data rate values to use in which formula. For schedulability analysis, the maximum possible execution rate of each SPI process has to be determined. This can be done by assuming maximum token production ($s_{i,max}$) and minimum token consumption ($r_{i,min}$) for the derivation of the RBE rate specifications, since these assumptions maximize the ratio of

$$\frac{x_{i+1}}{y_{i+1}} = \frac{s_i \cdot x_i}{r_i \cdot y_i}$$

which denotes the average process executions per time unit.

The imposed latency is independent of the data rate values, such that the bounds derived in [35] can be used. The inherent latency and, in particular, the determination of the number of additional required source process executions $N_0$ depends on the data rate values. When applying the algorithm to determine $N_0$ to SPI representations, the upper [lower] bound on $N_0$ can be obtained by assuming minimum [maximum] token production and maximum [minimum] token consumption. This can be motivated informally for the upper bound $N_{0,max}$ as follows. By producing the minimum number of tokens per execution while at the same time requiring the maximum number of tokens for activation

[1] The causal dependency between tokens expresses a possible value dependency between them and can be defined recursively as follows. A token $v$ is said to be causally dependent on another token $u$, if $v$ was produced by a process execution which either read $u$ or a token which is causally dependent on $u$.

of the consecutive process, the number of executions of the writing process in order to activate the reading process is maximized. Thus, more source process executions are needed to activate the sink process.

Regarding the second problem, the latency definitions of SPI and [35] are incompatible in general as the output of the causally dependent tokens may be performed by several executions of the output process (last process in chain before sink process) which even under the strong synchrony assumption are possibly not performed at the same time. For example, considering a chain of three processes where the source process $P_0$ ($R_0 = (1, y)$) produces 3 tokens per execution on queue $Q_0$ and process $P_1$ (which is also the output process) consumes 2 tokens from $Q_0$ at each execution. Then on the one hand, for every execution of $P_0$ there is an immediate execution of $P_1$ depending on the data tokens produced by $P_0$. Thus after [35], the inherent latency is 0. On the other hand, the first execution of $P_0$ (at time $t = 0$, no prior token on $Q_0$) produces a token (the third one) that is not consumed by the first execution of $P_1$ but by the second execution of $P_1$ which needs another token from the second execution of $P_0$ and thus executes at time $t = y$. Thus after the latency definition of the SPI model, the latency for the first production of three tokens is bounded by $[0, y]$ with $0$ originating from the first token and $y$ from the third token.

This incompatibility between both latency definitions does only occur when the larger of production and consumption rate value for a queue is not an integer multiple of the other. In this incompatibility case, it seems that the addition of an offset to the upper bound of the number of required source process executions ($N^{SPI}_{0,max} = N_{0,max} + 1$) is sufficient to obtain conservative latency bounds. A consequence of such an offset is that the use of minimum production ($s_{i,min}$) and maximum consumption ($r_{i,max}$) values for the determination of $N_{0,max}$ may no longer be conservative. This is due to the fact that an integer ratio between $s_{i,min}$ and $r_{i,max}$ may hide a potential non-integer ratio of values from the intervals leading to a larger value for $N_{0,max}$. The modification of the formula for $N_0$, however, is still work in progress.

# Chapter 7

# Summary and Outlook

An important trend in embedded system design is the growing system complexity. Fueled by increasing device integration capabilities of semiconductor technology, more functions can be implemented at the same cost. Besides the increase of the problem size, the growing complexity has another dimension which is the resulting heterogeneity with respect to the different functions and components of an embedded system. This means that functions from different application domains are tightly coupled in a single embedded system. As it is established industry practice that specialized specification languages and design environments are used in each application domain, the heterogeneity of embedded system functionality results in a heterogeneous, multi-language system specification.

The key problems in the context of multi-language design are the safe integration of the differently specified subsystems and the optimized implementation of the whole system. Both require the reliable validation of the system function as well as of the non-functional system properties. Current multi-language design approaches can be classified into cosimulational and compositional approaches.

Cosimulational approaches provide a flexible and systematic integration for heterogeneously specified systems. The two key strengths of cosimulational approaches are the generality to include all kinds of system parts as long as they can be executed or simulated as well as the reuse of the language-specific design environments and the resulting support of domain-specific optimization and analysis. Due to the fairly low-level integration of the different system parts, however, cosimulational approaches are less powerful for system-wide optimization and the validation of non-functional system properties.

Compositional approaches provide a deeper integration of the different system parts by creating a coherent formalism for the representation of the complete system at a higher level of abstraction. This homogeneous representation serves as a basis for system-wide analysis and optimization. This deep integration, however, compromises the flexibility as compositional approaches are typically restricted to a limited set of supported languages. Particularly problematic is the integration of reused components (e. g., legacy code) or incomplete specifications.

In this dissertation, a novel approach to the design of heterogeneously specified, complex embedded systems has been proposed. By assuming a truly heterogeneous multi-language specification while nevertheless providing an abstract homogeneous design representation supporting system-wide analysis and optimization, this approach combines the advantages of both cosimulational and compositional approaches. As it is based on

a homogeneous internal design representation, the presented approach is nevertheless a compositional approach.

This approach is called SPI workbench and focuses on the validation of non-functional system properties while abstracting the system functionality. In this sense, the SPI workbench rather augments existing cosimulational design flows with formal analysis capabilities regarding non-functional system properties than being a complete design environment.

The enabling technology for the system-wide analysis and optimization is the abstract internal system representation, the SPI model, which is obtained by extracting relevant properties from the multi-language specification. In this context, a crucial point for the applicability of the SPI workbench is the availability of transformations from input languages to the SPI model.

The SPI model is based on processes communicating via unidirectional channels having either FIFO queue or register semantics. The SPI model elements are characterized by a set of parameters instead of their detailed functionality. The parameters capture non-functional properties of the element such as timing, power consumption, and activation conditions. A major step towards high semantic flexibility of the model is the use of behavioral intervals for the model parameters. This allows the specification of incomplete information and thus facilitates the integration of system parts whose internal functional details are only partially known, such as legacy code. Due to the abstraction of functionality and the behavioral intervals, SPI is a non-executable model. Rather, a SPI representation of a system bounds all possible system behaviors with respect to its non-functional properties. While behavioral intervals allow the abstraction and clustering of different process execution behaviors, process modes support their explicit, distinct specification. Using both concepts, the degree of abstraction in a SPI representation can be controlled, allowing to effectively cope with system complexity.

Although being created to provide a homogeneous representation for heterogeneously specified systems, its variable level of abstraction makes SPI an appealing vehicle for the analysis of single-language systems as well. Another advantage of the SPI approach is that due to the abstraction only the structural implementation details have to be disclosed but all functional implementation details can be hidden. This ”grey box” view is particularly useful when validating performance for systems that are designed in a multi-company effort. An example for this is the design of automotive electronics where the car manufacturer and its suppliers have to commonly validate the system without giving away too many implementation details which may hurt their competing business interests.

Possible extensions of the SPI model include the representation of stochastic distributions for SPI parameters and dynamic process instantiation. Currently, the SPI model is targeted at the validation of hard constraints imposing a certain bound on a parameter. If this bound is violated, the system implementation is invalid. By providing a possibility to specify stochastic distributions for SPI parameters on top of the existing behavioral intervals, SPI would be extended to support quality of service analysis which are often used for multimedia or telecommunication applications. The representation of dynamic process instantiation would accommodate the increasing software share of embedded systems relying on this feature. Currently, different possible ways to represent dynamic process instantiation are evaluated. In [44] a possible modeling with the current set of model elements is presented.

The focus of this dissertation has been on the internal design representation, the SPI model, and its features. While the SPI model itself has been elaborated quite well, the surrounding methods and design steps remain work in progress. Current developments include heterogeneous analysis methods [84, 91] as well as systematic integration of different input languages.

# Appendix A

# SPIML

## A.1 Motivation

For the communication of SPI representations between different tools of the SPI workbench, which have been implemented in different programming languages, for different operating systems, or are used in a distributed environment, an common exchange format is needed. This exchange format may also be used to save SPI representations. Thus, requirements for the exchange format are human readability as well as tool support (e. g., availability of parsers etc.). For the exchange of structured data in text format, XML (eXtensible Markup Language) is established and excellently supported by a wide variety of public domain tools such as parsers, syntax checkers etc. Thus, the exchange format for SPI representations, called SPIML (SPI Markup Language), has been defined based on XML.

In this section, the main differences of SPIML to the SPI model as it has been defined in this work are motivated and explained. The probably most significant difference is the restriction to functional i. e. implementation independent parameters. This has been motivated in Section 4.3.2 with the definition of a matching architecture model and the placement of parameters, which depend on both application and target architecture, to mapping edges beetween elements of both models. Furthermore, only constraints concerning timing are currently represented in SPIML. This is due to the current focus of the SPI project on validation of system-level timing. However, SPIML can be easily extended to also cover, e. g., power consumption constraints.

A concept which previously has not been defined in this work is the port construct for processes and channels. The introduction of ports does not change the model semantics. Rather, it is an implementation decision that facilitates modularity and reuse of model elements as process parameters such as data rates no longer refer to the connected channel but to the process port the channel is connected to. Similarly motivated is the specification of token size as a parameter of ports instead of channels. In this case, it has to be checked if the token size parameters of ports connected via a common channel are equivalent.

In the following, the document type definition of SPIML version 1.0 is shown.

## A.2 Document Type Definition

```
<?xml version='1.0' encoding='utf-8' ?>
<!-- DTD of the SPI model. V 1.0, 31-Oct-2001 -->

<!-- ELEMENT Graph -->

<!ELEMENT Graph ( ( Process |
                    Queue |
                    Register |
                    Edge |
                    Interface )* ,
                  Constraints? ,
                  Comment? ,
                  Auxil? )>

<!ATTLIST Graph  name CDATA  #IMPLIED >


<!-- ELEMENT Process -->

<!ELEMENT Process ( ( Input |
                      Output )+ ,
                    Mode+ ,
                    ActivationRule+ ,
                    TagProductionRule* ,
                    Predicate+ ,
                    Comment? ,
                    Auxil? )>

<!ATTLIST Process  name    CDATA  #IMPLIED
                   id      ID     #REQUIRED
                   virtual  (yes | no )  'no' >


<!-- ELEMENT Mode -->

<!ELEMENT Mode ( Rate+ ,
                 Comment? ,
                 Auxil? )>

<!ATTLIST Mode  name               CDATA  #REQUIRED
                tagproductionrules CDATA  #IMPLIED >


<!-- ELEMENT Rate -->

<!ELEMENT Rate ( Comment? ,
                 Auxil? )>

<!ATTLIST Rate  port CDATA  #REQUIRED
                low  CDATA  #REQUIRED
                high CDATA  #REQUIRED >
```

```
<!-- ELEMENT ActivationRule -->

<!ELEMENT ActivationRule ( Comment?,
                           Auxil? )>

<!ATTLIST ActivationRule  predicate CDATA  #REQUIRED
                          modes     CDATA  #REQUIRED >


<!-- ELEMENT TagProductionRule -->

<!ELEMENT TagProductionRule ( Comment? ,
                              Auxil? )>

<!ATTLIST TagProductionRule  name       CDATA  #REQUIRED
                             predicate  CDATA  #REQUIRED
                             production CDATA  #REQUIRED >


<!-- ELEMENT Predicate -->

<!ELEMENT Predicate ( Comment? ,
                      Auxil? )>

<!ATTLIST Predicate  name            CDATA  #REQUIRED
                     predicatestring CDATA  #REQUIRED >


<!-- ELEMENT Queue -->

<!ELEMENT Queue ( Input ,
                  Output ,
                  Token* ,
                  Comment? ,
                  Auxil? )>

<!ATTLIST Queue  name    CDATA  #IMPLIED
                 id      ID     #REQUIRED
                 virtual ( yes | no ) 'no' >


<!-- ELEMENT Register -->

<!ELEMENT Register ( Input ,
                     Output ,
                     Token? ,
                     Comment? ,
                     Auxil? )>

<!ATTLIST Register  name    CDATA  #IMPLIED
                    id      ID     #REQUIRED
                    virtual ( yes | no ) 'no' >


<!-- ELEMENT Input, Output -->
```

```
<!ELEMENT Input ( Tokensize? ,
                  Comment? ,
                  Auxil? )>

<!ATTLIST Input  name       CDATA  #REQUIRED
                 id         ID     #REQUIRED
                 commregion CDATA  #IMPLIED >

<!ELEMENT Output ( Tokensize? ,
                   Comment? ,
                   Auxil? )>

<!ATTLIST Output  name       CDATA  #REQUIRED
                  id         ID     #REQUIRED
                  commregion CDATA  #IMPLIED >


<!-- ELEMENT Tokensize -->

<!ELEMENT Tokensize ( Comment? ,
                      Auxil? )>

<!ATTLIST Tokensize  low  CDATA  #REQUIRED
                     high CDATA  #REQUIRED >



<!-- ELEMENT Token -->

<!ELEMENT Token ( Comment? ,
                  Auxil? )>

<!ATTLIST Token  pos  CDATA  #REQUIRED
                 tags CDATA  #IMPLIED >


<!-- ELEMENT Edge -->

<!ELEMENT Edge ( Comment? ,
                 Auxil? )>

<!ATTLIST Edge  source IDREF  #REQUIRED
                target IDREF  #REQUIRED >


<!-- ELEMENT Interface -->

<!ELEMENT Interface ( ( Input |
                        Output )+ ,
                      InterfaceConfiguration* ,
                      ConfigurationSelectionRule* ,
                      Predicate* ,
                      Comment? ,
```

```
                      Auxil? )>

<!ATTLIST Interface  name CDATA  #IMPLIED
                     id   ID     #REQUIRED >


<!-- ELEMENT InterfaceConfiguration -->

<!ELEMENT InterfaceConfiguration ( Cluster ,
                                   PortMapping+ ,
                                   Comment? ,
                                   Auxil? )>

<!ATTLIST InterfaceConfiguration  name    CDATA  #REQUIRED >


<!-- ELEMENT ConfigurationSelectionRule -->

<!ELEMENT ConfigurationSelectionRule ( Comment? ,
                                       Auxil? )>

<!ATTLIST ConfigurationSelectionRule  predicate      CDATA  #REQUIRED
                                      configurations CDATA  #REQUIRED >


<!-- ELEMENT Cluster -->

<!ELEMENT Cluster ( ( Input |
                      Output )+ ,
                    PortMapping+ ,
                    Graph ,
                    Comment? ,
                    Auxil? )>

<!ATTLIST Cluster  name CDATA  #IMPLIED >


<!-- ELEMENT PortMapping -->

<!ELEMENT PortMapping ( Comment? ,
                        Auxil? )>

<!ATTLIST PortMapping  source IDREF  #REQUIRED
                       target IDREF  #REQUIRED >


<!-- ELEMENT Constraints -->

<!ELEMENT Constraints ( LatencyPathConstraint* ,
                        Comment? ,
                        Auxil? )>


<!-- ELEMENT LatencyPathConstraint -->
```

```
<!ELEMENT LatencyPathConstraint ( Latency ,
                                  Comment? ,
                                  Auxil? )>

<!ATTLIST LatencyPathConstraint  name CDATA   #IMPLIED
                                 path IDREFS  #REQUIRED >


<!-- ELEMENT Latency -->

<!ELEMENT Latency ( Comment? ,
                    Auxil? )>

<!ATTLIST Latency  low  CDATA  #REQUIRED
                   high CDATA  #REQUIRED >


<!-- ELEMENT Comment -->

<!ELEMENT Comment ( #PCDATA )>


<!-- ELEMENT Auxil -->

<!ELEMENT Auxil ANY>
```

# Bibliography


[1] Arexsys. ArchiMate data sheet. http://www.arexsys.com, 2002.

[2] F. Balarin, P. Giusto, A. Jurecska, et al. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, May 1997.

[3] Th. Benner and R. Ernst. An approach to mixed systems co-synthesis. In *Proceedings Fifth International Workshop on Hardware/Software Co-Design (Codes/CASHE '97)*, pages 9–14, 1997.

[4] G. Berry, P. Couronne, and G. Gonthier. The synchronous approach to reactive and real-time systems. *IEEE Proceedings*, 79, 1991.

[5] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2), February 1996.

[6] J. Blazewicz. *Modeling and Performance Evaluation of Computer Systems*, chapter Scheduling Dependent Tasks with Different Arrival Times to Meet Deadlines. North-Holland, Amsterdam, 1976.

[7] R. Braek and O. Haugen. *Engineering Real Time Systems. An Object-Oriented Methodology using SDL*. Prentice-Hall, Inc., 1993.

[8] Joseph Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model*. PhD thesis, University of California at Berkeley, 1993.

[9] Joseph Buck and Radha Vaidyanathan. Heterogeneous modeling and simulation of embedded systems in El Greco. In *Proceedings Eightth International Workshop on Hardware/Software Co-Design (Codes/CASHE '00)*, San Diego, California, May 2000.

[10] Cadence. Signal processing workbench (SPW). http://www.cadence.com/, 2002.

[11] Cadence. Virtual component co-design (VCC) environment. White Paper, http://www.cadence.com/systems, 2002.

[12] Cadence. Virtual component co-design (VCC) links to implementation. White Paper. http://www.cadence.com/systems, 2002.

[13] Ying Cai, Marek Jersak, Kai Richter, Zhang Yu, Dirk Ziegenbein, and Rolf Ernst. Bericht zum DFG-Projekt "Kombination von Sprachen und Berechnungsmodellen für die Synthese eingebetteter Systeme" (project report "combination of languages and models of computation for embedded system synthesis"; in German), 2001.

[14] H. Chang, L. Cooke, M.Hunt, et al. *Surviving the SoC Revolution*. Kluwer Academic Publishers, 1999.

[15] M. Chiodo, P. Giusto, H. Hsieh, et al. A formal methodology for hardware/software codesign of embedded systems. *IEEE Micro*, August 1994.

[16] P. Chou, K. Hines, K. Partridge, and G. Boriello. Control generation for embedded systems based on composition of modal processes. In *Proceedings International Conference on Computer-Aided Design (ICCAD '98)*, pages 46–53, San Jose, CA, USA, November 1998.

[17] P. Chou, R. B. Ortega, and G. Borriello. The Chinook hardware/software co-synthesis system. In *Proceedings 8th International Symposium on System Synthesis (ISSS '95)*, pages 22–27, Cannes, France, September 1995.

[18] F. Commoner and A. Holt. Marked directed graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.

[19] P. Coste, F. Hessel, Ph. Le Marrec, et al. Multilanguage design of heterogeneous systems. In *Proceedings Seventh International Workshop on Hardware/Software Co-Design (Codes/CASHE '99)*, pages 54–58, Rome, Italy, May 1999.

[20] CoWare. N2C data sheet. http://www.coware.com, 2002.

[21] A. Dasdan. *Timing Analysis of Embedded Real-Time Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[22] John Davis, Christopher Hylands, Jörn Janneck, Edward A. Lee, et al. Overview of the Ptolemy project. Technical Report UCB/ERL M01/11, University of California at Berkeley, March 2001.

[23] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, et al. YAPI: Application modelling for signal processing systems. In *Proceedings 37th Design Automation Conference (DAC '00)*, Los Angeles, California, June 2000.

[24] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: Task graphs for free. In *Proceedings Sixth International Workshop on Hardware/Software Co-Design (Codes/CASHE '98)*, page 101, Seattle, USA, March 1998.

[25] VSI Alliance System-Level Design DWG. VSI alliance model taxonomy version 2.1. http://www.vsi.org, July 2001.

[26] R. Ernst and Th. Benner. Communication, constraints and user directives in COSYMA. Technical Report CY-94-2, Institut für DV-Anlagen, Technische Universität Braunschweig, 1994.

[27] Rolf Ernst. *System-Level Synthesis*, volume 357 of *NATO Science Series E: Applied Sciences*, chapter Embedded System Architectures. Kluwer Academic Publishers, 1999.

[28] ETAS. ERCOS$^{ek}$, OSEK based real-time operating system. http://www.etas.de/, 2002.

[29] C. J. Fidge. Real-time schedulability tests for preemptive multitasking. *Journal of Real-Time Systems*, 14:61–93, 1998.

[30] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice-Hall, Inc., 1994.

[31] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[32] R. Gerber, W. Pugh, and M. Saksena. Parametric dispatching of hard real-time taks. *IEEE Transaction on Computers*, 44(3):471–479, March 1995.

[33] A. Gerstlauer, R. Dömer, J. Peng, and D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2000.

[34] A. Girault, B. Lee, and E. A. Lee. A preliminary study of hierarchical finite state machines with multiple concurrency models. Technical Report UCB/ERL M97/57, Electronics Research Laboratory, College of Engineering, Univ. of Cal. at Berkeley, 1997.

[35] Steve Goddard. *On the Management of Latency in the Synthesis of Real-Time Signal Processing Systems from Processing Graphs*. PhD thesis, University of North Carolina at Chapel Hill, 1998.

[36] Steve Goddard and Kevin Jeffay. Analyzing the real-time properties of a dataflow execution paradigm using a synthetic aperture radar application. In *Proceedings IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, pages 60–71, Montreal, Canada, June 1997.

[37] Mentor Graphics. DSP Station data sheet. http://www.mentor.com/, 1999.

[38] Mentor Graphics. Seamless CVE data sheet. http://www.mentor.com/seamless, 2002.

[39] T. Grötker, R. Schoenen, and H. Meyr. PCC: A modeling technique for mixed control/data flow systems. In *Proceedings European Design & Test Conference (ED&TC '97)*, pages 482–486, Paris, France, March 1997.

[40] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[41] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[42] David Harel. StateCharts: A visual formalism for complex systems. *Science of Computer Programming*, 1987.

[43] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, and other. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[44] Rafik Henia. Entwicklung eines Prototypen zur Übersetzung von SDL nach SPI (development of a prototype for the translation of sdl to spi; in German). Technical report, Institute for Computer and Communication Network Engineering, Technical University of Braunschweig, Germany, 2001.

[45] Frank Heuschen and Klaus Waldschmidt. Analog/digital co-design: System level analysis for signal processing embedded systems. In *Proceedings SASIMI 2001*, pages 402–409, Nara, Japan, October 2001.

[46] Harry Hsieh, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. Embedded system co-design: Synthesis and verification. In *Proceedings NATO Workshop on HW/SW CO-Design*, Italy, June 1995.

[47] Open SystemC Initiative. SystemC. http://www.systemc.org/, 2002.

[48] SLDL Initiative. Rosetta usage and semantics guides. http:/www.sldl.org/, 2002.

[49] ITU-T. Recommendation Z.100. CCITT specification and description language SDL, 1993.

[50] ITU-T. Z.120. message sequence chart, 1996.

[51] Kevin Jeffay and Steve Goddard. A theory of rate-based execution. In *Proceedings 20th IEEE Real-Time Systems Symposium (RTSS '99)*, pages 304–314, Phoenix, Arizona, December 1999.

[52] Kurt Jensen. *Colored Petri Nets - Basic Concepts, Analysis Methods and Practical Use, Vol. 1*. Springer Verlag, 1992.

[53] A.A. Jerraya and K. O'Brien. *Computer Aided Software/Hardware Engineering*, chapter SOLAR: An Intermediate Format for System-Level Modeling and Synthesis. IEEE Press, 1994.

[54] A.A. Jerraya, M. Romdhani, Ph. Le Marrec, F. Hessel, et al. *System-Level Synthesis*, volume 357 of *NATO Science Series E: Applied Sciences*, chapter Multilanguage Specification for System Design. Kluwer Academic Publishers, 1999.

[55] M. Jersak, D. Ziegenbein, and R. Ernst. A general approach to modeling system-level timing constraints. In *Accepted for publication - Proceedings 4th Forum on Design Languages*, Lyon, France, 2001.

[56] Marek Jersak, Ying Cai, and Amilcar Lucas. Translating Simulink to SPI. Technical Report TR-SPI-01-02, Institute of Computer and Communication Networks Engineering, Technical University of Braunschweig, Germany, 2001.

[57] Marek Jersak, Ying Cai, Dirk Ziegenbein, and Rolf Ernst. A transformational approach to constraint relaxation of a time-driven simulation model. In *Proceedings 13th International Symposium on System Synthesis*, Madrid, Spain, September 2000.

[58] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*, pages 471–475. North Holland, 1974.

[59] Gilles Kahn and D. MacQueen. Coroutines and networks of parallel processes. In *Proceedings of the IFIP Congress 77*. North Holland, 1977.

[60] R. Karp and R. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal*, 14:1390–1411, November 1966.

[61] P. V. Knudsen and J. Madsen. Communication estimation for hardware/software codesign. In *Proceedings Sixth International Workshop on Hardware/Software Co-Design (Codes/CASHE '98)*, pages 55–59, Seattle, WA, USA, March 1998.

[62] H. Kopetz and G. Gruensteidl. TTP - a time-triggered protocol for fault-tolerant computin g. In *Proceedings 23rd International Symposium on Fault-Tolerant Co mputing*, pages 524–532, 1993.

[63] D. Ku and G. De Micheli. HardwareC: a language for hardware design. Technical Report SCSL/CSL/TR-90-419, Computer Systems Laboratory, Stanford University, Stanford, California, August 1990.

[64] Alexander Kuenz. An architectural model for heterogeneous hardware/software platforms. Master's thesis, Technical University of Braunschweig, Germany, 2001.

[65] Naval Research Laboratory. Processing graph method specification, 1987.

[66] Luciano Lavagno, Alberto Sangiovanni-Vincentelli, and Ellen Sentovich. *Models of Computation for Embedded System Design*, volume 357 of *NATO Science Series E: Applied Sciences*, chapter Embedded System Architectures. Kluwer Academic Publishers, 1999.

[67] E. A. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1), January 1987.

[68] E. A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[69] E. A. Lee and Th. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995.

[70] Y. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[71] C. Liu and J. Layland. Scheduling algorithm for multiprogramming in a hard-real-time environment. *Journal of the ACM*, pages 46–61, 1973.

[72] F. Maraninchi. Argonaute: Graphical description, semantics, and verification of reactive systems by using a process algebra. In *Proceedings International Workshop Automatic Verification Methods for Finite State Machines*, New York, 1989.

[73] The MathWorks. Real-Time Workshop user's guide version 3, January 1999.

[74] The MathWorks. Using Simulink version 3, January 1999.

[75] The MathWorks. Stateflow data sheet. http://www.mathworks.com/, 2002.

[76] A. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, MIT, May 1983.

[77] Jean-Pierre Moreau, Philippe di Crescenzo, and Lloyd Pople. Hardware/software system codesign based on SDL/C specifications. *Microelectronic Engineering, Elsevier Science*, 2001.

[78] P. Panagaden and V. Shanbhogue. The expressive power of indeterminate dataflow primitives. *Information and Computation*, 98(1), May 1992.

[79] C. Petri. Interpretations of a net theory. Technical Report 75-07, GMD, Bonn, Germany, 1975.

[80] K. Richter. Developing a general model for scheduling of mixed transformative/reactive systems. Master's thesis, Technical University of Braunschweig, January 1998.

[81] Kai Richter and Rolf Ernst. Event model interfaces for heterogeneous system analysis. In *Proceedings of Design Automation and Test in Europe (DATE '02)*, Paris, France, March 2002.

[82] Kai Richter, Rolf Ernst, and Wayne Wolf. Hierarchical specification methods for platform-based design. In *Proceedings SASIMI 2001*, pages 25–30, Nara, Japan, 2001.

[83] Kai Richter, Dirk Ziegenbein, Rolf Ernst, Lothar Thiele, and Jürgen Teich. Representation of function variants for embedded system optimizetion and systhesis. In *Proceeding 36th Design Automation Conference*, pages 517–522, New Orleans, USA, June 1999.

[84] Kai Richter, Dirk Ziegenbein, Marek Jersak, Fabian Wolf, and Rolf Ernst. Model composition for scheduling analysis and platform design. In *submitted to DAC02*, 2002.

[85] M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal models for embedded system design. *IEEE Design & Test of Computers*, 17(2):14–27, 2000.

[86] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of the IEEE*, 82(1):68–82, January 1994.

[87] Karsten Strehl. *Symbolic Methods Applied to Formal Verification and Synthesis in Embedded System Design*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 2000.

[88] Karsten Strehl, Lothar Thiele, Matthias Gries, Dirk Ziegenbein, Rolf Ernst, and Jürgen Teich. FunState - an internal design representation for codesign. *IEEE Transaction on Very Large Scale Integrated (VLSI) Systems*, 9(4):524–544, August 2001.

[89] Synopsys. CoCentric System Studio reference guide, August 2000.

[90] Synopsys. COSSAP. http://www.synopsys.com/products/dsp/dsp.html, 2002.

[91] Lothar Thiele, Samarjit Chakraborty, Matthias Gries, Alexander Maxiaguine, and Jonas Greutert. Embedded software in network processors - models and algorithms. In *First Workshop on Embedded Software (EMSOFT)*, Lecture Notes in Computer Science 2211, pages 416–434, Lake Tahoe, CA, USA, October 2001. Springer.

[92] Jan Uerpmann. Studie zum Scheduling von SPI-Systemen auf der Basis von Echtzeitbetriebssystemen (study of scheduling for SPI-systems based on real-time operating systems; in German). Master's thesis, Technical University of Braunschweig, 2000.

[93] C. A. Valderama, M. Romdhani, J. M. Daveau, et al. *Hardware/Software Co-Design: Prinicples and Practice*, chapter COSMOS: A Transformational Co-design Tool for Multiprocessor Architectures. Kluwer Academic Publishers, 1997.

[94] C. A. Valderrama, A. Changuel, P.V. Raghavan, M. Abid, T. Ben Ismail, and A. A. Jerraya. A unified model for co-simulation and co-synthesis of mixed hardware/software systems. In *Proceedings European Design & Test Conference (ED&TC '95)*, Paris, France, March 1995.

[95] F. Wolf and R. Ernst. Execution cost interval refinement in static software analysis. *Journal of Systems Architecture, The EUROMICRO Journal, Special Issue on Modern Methods and Tools in Digital System Design*, 47(3-4):339–356, April 2001.

[96] Fabian Wolf. *Behavioral Intervals in Embedded System Design*. PhD thesis, Technical University of Braunschweig, Germany, 2001.

[97] D. Ziegenbein, J. Uerpmann, and R. Ernst. Dynamic Response Time Optimization for SDF Graphs. In *Proceedings International Conference on Computer-Aided Design (ICCAD '00)*, San Jose, November 2000.

[98] D. Ziegenbein, F. Wolf, K. Richter, M. Jersak, and R. Ernst. Interval-based analysis of software processes. In *Proceedings Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '2001)*, Snowbird, USA, June 2001.

[99] W. M. Zuberek. Timed Petri nets: Definitions, propertiesm and applications. *Microelectronics and Reliability*, 31(4):627–644, 1991.

# Nomenclature

| | |
|---|---|
| ANSI | American National Standards Institute |
| $b_c$ | SPI: size of each token communicated on channel $c$ |
| BDF | Boolean dataflow |
| $C$ | SPI: set of channels |
| $cr_c$ | SPI: communication region of a process for channel $c$ |
| CFSM | Codesign Finite State Machine |
| CSDF | cyclo-static dataflow |
| $D(tag)$ | SPI: finite value domain of mode tag $tag$ |
| $d_c$ | SPI: initial number of data tokens on channel $c$ |
| DDF | dynamic dataflow |
| DF | dataflow |
| $E$ | SPI: set of edges |
| EDF | earliest deadline first |
| FFT | Fast Fourier Transformation |
| FIFO | First-In-First-Out |
| FSM | finite state machine |
| $\Gamma$ | SPI: set of clusters |
| $\gamma$ | SPI: cluster |
| GRMS | generalized rate monotonic analysis |
| HCFSM | hierarchical concurrent finite state machine |
| HW | hardware |
| $Inputs_p$ | SPI: set of input channels of process $p$ |

| | |
|---|---|
| ILP | integer linear programming |
| IP | intellectual property |
| $lat_p$ | SPI: latency time interval of process $p$ |
| $LC$ | SPI: latency path constraint |
| $M_p$ | SPI: process mode set of process $p$ |
| $m_p$ | SPI: process mode of process $p$ |
| $mem_p$ | SPI: static and dynamic memory size intervals of process $p$ |
| MOC | model of computation |
| $Outputs_p$ | SPI: set of output channels of process $p$ |
| $\Psi$ | SPI: set of interfaces |
| $\psi$ | SPI: interface |
| $P$ | SPI: set of processes |
| $PC$ | SPI: power consumption constraint |
| $pow_p$ | SPI: power consumption interval of process $p$ |
| PCC | Process Coordination Calculus |
| PGM | processing graph method |
| $Q$ | SPI: set of queues |
| $R$ | SPI: set of registers |
| $r_c$ | SPI: input data rate interval bounding the number of tokens a process reads from channel $c$ |
| RBE | rate-based execution |
| RMS | rate monotonic analysis |
| RPC | remote procedure call |
| RTL | register transfer level |
| RTW | Real-Time Workshop |
| $s_c$ | SPI: input data rate interval bounding the number of tokens a process writes to channel $c$ |
| SDF | synchronous dataflow |
| SDL | Specification and Description Language |

| | |
|---|---|
| SPI | system property intervals |
| SPIML | SPI Markup Language |
| SW | software |
| SYMTA | SYMbolic Timing Analysis tool suite |
| $TP_p$ | SPI: set of mode tag production rules of process $p$ |
| $TS_a$ | SPI: mode tag set of a token $a$ |
| $v_n$ | SPI: virtuality flag (true if model element $n$ is virtual) |
| VHDL | VHSIC hardware description language |
| VHSIC | very high speed integrated circuit |
| VSIA | Virtual Socket Interface Alliance |
| XML | eXtensible Markup Language |